Abstract

This document is intended for knowledgeable users of C (or any other language using a C-like grammar, like Perl or Java) who would like to know more about, or make the transition to, C++. This document is the main textbook for Frank’s C++ programming courses, which are yearly organized at the University of Groningen. The C++ Annotations do not cover all aspects of C++, though. In particular, C++’s basic grammar is not covered when equal to C’s grammar. Any basic book on C may be consulted to refresh that part of C++’s grammar.

If you want a hard-copy version of the C++ Annotations: printable versions are available in postscript, pdf and other formats in

http://sourceforge.net/projects/cppannotations/,

in files having names starting with cplusplus (A4 paper size). Files having names starting with ‘cplusplusus’ are intended for the US legal paper size. The C++ Annotations are also available as a Kindle book.

The latest version of the C++ Annotations in html-format can be browsed at:

http://cppannotations.sourceforge.net/

and/or at

http://www.icce.rug.nl/documents/
Contents

1 Overview Of The Chapters 1

2 Introduction 3

2.1 What’s new in the C++ Annotations .............................................. 4

2.2 C++’s history ................................................................................. 7

2.2.1 History of the C++ Annotations ................................................. 7

2.2.2 Compiling a C program using a C++ compiler ......................... 7

2.2.3 Compiling a C++ program .......................................................... 8

2.3 C++: advantages and claims .......................................................... 10

2.4 What is Object-Oriented Programming? ....................................... 11

2.5 Differences between C and C++ ................................................... 12

2.5.1 The function ‘main’ ................................................................. 13

2.5.2 End-of-line comment .............................................................. 13

2.5.3 Strict type checking ................................................................. 13

2.5.4 Function Overloading .............................................................. 14

2.5.5 Default function arguments .................................................... 15

2.5.6 NULL-pointers vs. 0-pointers and nullptr ............................... 16

2.5.7 The ‘void’ parameter list ......................................................... 17

2.5.8 The ‘#define __cplusplus’ ....................................................... 17

2.5.9 Using standard C functions .................................................... 18

2.5.10 Header files for both C and C++ .............................................. 18

2.5.11 Defining local variables ......................................................... 19

2.5.12 The keyword ‘typedef’ ......................................................... 22

2.5.13 Functions as part of a struct .................................................. 22
# A First Impression Of C++

## 3.1 Extensions to C

- **3.1.1 Namespaces**
- **3.1.2 The scope resolution operator ::**
- **3.1.3 Using the keyword 'const'**
- **3.1.4 'cout', 'cin', and 'cerr'**

## 3.2 Functions as part of structs

- **3.2.1 Data hiding: public, private and class**
- **3.2.2 Structs in C vs. structs in C++**

## 3.3 More extensions to C

- **3.3.1 References**
- **3.3.2 Rvalue References**
- **3.3.3 Strongly typed enumerations**
- **3.3.4 Initializer lists**
- **3.3.5 Type inference using 'auto'**
- **3.3.6 Defining types and 'using' declarations**
- **3.3.7 Range-based for-loops**
- **3.3.8 Raw String Literals**
- **3.3.9 Binary constants**

## 3.4 New language-defined data types

- **3.4.1 The data type 'bool'**
- **3.4.2 The data type 'wchar_t'**
- **3.4.3 Unicode encoding**
- **3.4.4 The data type 'long long int'**
- **3.4.5 The data type 'size_t'**

## 3.5 A new syntax for casts

- **3.5.1 The 'static_cast'-operator**
- **3.5.2 The 'const_cast'-operator**
- **3.5.3 The 'reinterpret_cast'-operator**
- **3.5.4 The 'dynamic_cast'-operator**
- **3.5.5 Casting 'shared_ptr' objects**

## 3.6 Keywords and reserved names in C++
4 Name Spaces

4.1 Namespaces ......................................................... 57
    4.1.1 Defining namespaces ........................................ 57
    4.1.2 Referring to entities ........................................ 59
    4.1.3 The standard namespace ..................................... 63
    4.1.4 Nesting namespaces and namespace aliasing ............... 64

5 The ‘string’ Data Type ........................................... 69

5.1 Operations on strings ........................................... 70
5.2 A std::string reference ......................................... 71
    5.2.1 Initializers .................................................. 72
    5.2.2 Iterators ..................................................... 73
    5.2.3 Operators .................................................... 73
    5.2.4 Member functions .......................................... 74
    5.2.5 Conversion functions ...................................... 81

6 The IO-stream Library ............................................. 85

6.1 Special header files ........................................... 88
6.2 The foundation: the class ‘ios_base’ .......................... 89
6.3 Interfacing ‘streambuf’ objects: the class ‘ios’ .......... 89
    6.3.1 Condition states ............................................ 90
    6.3.2 Formatting output and input ............................... 93
6.4 Output ............................................................ 100
    6.4.1 Basic output: the class ‘ostream’ .......................... 100
    6.4.2 Output to files: the class ‘ofstream’ ..................... 103
    6.4.3 Output to memory: the class ‘ostringstream’ ......... 105
6.5 Input ............................................................. 106
    6.5.1 Basic input: the class ‘istream’ .......................... 107
    6.5.2 Input from files: the class ‘ifstream’ .................... 110
    6.5.3 Input from memory: the class ‘istringstream’ .......... 111
    6.5.4 Copying streams ............................................ 112
    6.5.5 Coupling streams .......................................... 113
6.6 Advanced topics ................................................ 114
8.1.2 Public static data ............................................. 166
8.1.3 Initializing static const data ................................. 166
8.1.4 Generalized constant expressions (constexpr) .......... 166
8.2 Static member functions ........................................... 170
  8.2.1 Calling conventions ........................................ 172

9 Classes And Memory Allocation ................................... 175
  9.1 Operators ‘new’ and ‘delete’ ............................ 176
    9.1.1 Allocating arrays ........................................ 177
    9.1.2 Deleting arrays .......................................... 178
    9.1.3 Enlarging arrays ......................................... 179
    9.1.4 Managing ‘raw’ memory ............................... 180
    9.1.5 The ‘placement new’ operator ....................... 180
  9.2 The destructor .................................................. 183
    9.2.1 Object pointers revisited ............................ 185
    9.2.2 The function set_new_handler() ................... 188
  9.3 The assignment operator ...................................... 189
    9.3.1 Overloading the assignment operator .............. 190
  9.4 The ‘this’ pointer .............................................. 194
    9.4.1 Sequential assignments and this .................. 194
  9.5 The copy constructor: initialization vs. assignment .... 195
  9.6 Revising the assignment operator ......................... 197
    9.6.1 Swapping .................................................. 198
  9.7 Moving data .................................................... 202
    9.7.1 The move constructor (dynamic data) ............... 204
    9.7.2 The move constructor (composition) .................. 206
    9.7.3 Move-assignment ....................................... 208
    9.7.4 Revising the assignment operator (part II) ......... 208
    9.7.5 Moving and the destructor ............................ 209
    9.7.6 Move-only classes ..................................... 210
    9.7.7 Default move constructors and assignment operators .... 210
    9.7.8 Moving: implications for class design ................ 212
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.8</td>
<td>Copy Elision and Return Value Optimization</td>
<td>212</td>
</tr>
<tr>
<td>9.9</td>
<td>Plain Old Data</td>
<td>214</td>
</tr>
<tr>
<td>9.10</td>
<td>Conclusion</td>
<td>215</td>
</tr>
<tr>
<td>10</td>
<td>Exceptions</td>
<td>217</td>
</tr>
<tr>
<td>10.1</td>
<td>Exception syntax</td>
<td>218</td>
</tr>
<tr>
<td>10.2</td>
<td>An example using exceptions</td>
<td>218</td>
</tr>
<tr>
<td>10.2.1</td>
<td>Anachronisms: ‘setjmp’ and ‘longjmp’</td>
<td>220</td>
</tr>
<tr>
<td>10.2.2</td>
<td>Exceptions: the preferred alternative</td>
<td>222</td>
</tr>
<tr>
<td>10.3</td>
<td>Throwing exceptions</td>
<td>223</td>
</tr>
<tr>
<td>10.3.1</td>
<td>The empty ‘throw’ statement</td>
<td>227</td>
</tr>
<tr>
<td>10.4</td>
<td>The try block</td>
<td>228</td>
</tr>
<tr>
<td>10.5</td>
<td>Catching exceptions</td>
<td>229</td>
</tr>
<tr>
<td>10.5.1</td>
<td>The default catcher</td>
<td>231</td>
</tr>
<tr>
<td>10.6</td>
<td>Declaring exception throwers (deprecated)</td>
<td>232</td>
</tr>
<tr>
<td>10.7</td>
<td>Iostreams and exceptions</td>
<td>235</td>
</tr>
<tr>
<td>10.8</td>
<td>Standard Exceptions</td>
<td>236</td>
</tr>
<tr>
<td>10.9</td>
<td>System error, error code and error category</td>
<td>237</td>
</tr>
<tr>
<td>10.9.1</td>
<td>The class ‘std::error_code’</td>
<td>238</td>
</tr>
<tr>
<td>10.9.2</td>
<td>The class ‘std::error_category’</td>
<td>239</td>
</tr>
<tr>
<td>10.10</td>
<td>Exception guarantees</td>
<td>241</td>
</tr>
<tr>
<td>10.10.1</td>
<td>The basic guarantee</td>
<td>242</td>
</tr>
<tr>
<td>10.10.2</td>
<td>The strong guarantee</td>
<td>243</td>
</tr>
<tr>
<td>10.10.3</td>
<td>The nothrow guarantee</td>
<td>245</td>
</tr>
<tr>
<td>10.11</td>
<td>Function try blocks</td>
<td>246</td>
</tr>
<tr>
<td>10.12</td>
<td>Exceptions in constructors and destructors</td>
<td>249</td>
</tr>
<tr>
<td>11</td>
<td>More Operator Overloading</td>
<td>257</td>
</tr>
<tr>
<td>11.1</td>
<td>Overloading ‘operator<a href=""></a>’</td>
<td>257</td>
</tr>
<tr>
<td>11.2</td>
<td>Overloading the insertion and extraction operators</td>
<td>260</td>
</tr>
<tr>
<td>11.3</td>
<td>Conversion operators</td>
<td>262</td>
</tr>
<tr>
<td>11.4</td>
<td>The keyword ‘explicit’</td>
<td>265</td>
</tr>
<tr>
<td>11.4.1</td>
<td>Explicit conversion operators</td>
<td>267</td>
</tr>
</tbody>
</table>
11.5 Overloading the increment and decrement operators .......................... 267
11.6 Overloading binary operators ...................................................... 269
11.7 Overloading 'operator new(size_t)' ............................................. 274
11.8 Overloading 'operator delete(void *)' ................................-------- 276
11.9 Operators 'new[]' and 'delete[]' ............................................... 277
  11.9.1 Overloading 'new[]' ....................................................... 277
  11.9.2 Overloading 'delete[]' .................................................... 278
  11.9.3 'new[]', 'delete[]' and exceptions ........................................ 280
11.10 Function Objects ................................................................. 282
  11.10.1 Constructing manipulators ............................................... 284
11.11 The case of [io]fstream::open() ............................................ 287
11.12 User-defined literals ............................................................. 289
11.13 Overloadable operators .......................................................... 290

12 Abstract Containers ..................................................................... 293
  12.1 Notations used in this chapter .................................................. 295
  12.2 The 'pair' container ............................................................... 296
  12.3 Allocators ............................................................................... 297
  12.4 Available Containers ............................................................... 298
    12.4.1 The 'array' container ....................................................... 298
    12.4.2 The 'vector' container ..................................................... 300
    12.4.3 The 'list' container ......................................................... 303
    12.4.4 The 'queue' container ..................................................... 310
    12.4.5 The 'priority_queue' container ......................................... 312
    12.4.6 The 'deque' container ...................................................... 314
    12.4.7 The 'map' container ......................................................... 317
    12.4.8 The 'multimap' container .................................................. 326
    12.4.9 The 'set' container .......................................................... 328
    12.4.10 The 'multiset' container ................................................. 331
    12.4.11 The 'stack' container ...................................................... 333
    12.4.12 The 'unordered_map' container ('hash table') ..................... 335
    12.4.13 The 'unordered_set' container ......................................... 343
14.5 Virtual functions and multiple inheritance .............................................. 391
   14.5.1 Ambiguity in multiple inheritance ................................................. 392
   14.5.2 Virtual base classes ......................................................................... 393
   14.5.3 When virtual derivation is not appropriate ........................................ 397
14.6 Run-time type identification ................................................................. 398
   14.6.1 The dynamic_cast operator .............................................................. 399
   14.6.2 The ‘typeid’ operator ..................................................................... 402
14.7 Inheritance: when to use to achieve what? ............................................. 404
14.8 The ‘streambuf’ class ............................................................................. 407
   14.8.1 Protected ‘streambuf’ members ......................................................... 409
   14.8.2 The class ‘filebuf’ ........................................................................... 414
14.9 A polymorphic exception class ............................................................... 415
14.10 How polymorphism is implemented ....................................................... 418
14.11 Undefined reference to vtable ............................................................... 421
14.12 Virtual constructors ............................................................................. 422

15 Friends ........................................................................................................ 427
   15.1 Friend functions .................................................................................. 428
   15.2 Extended friend declarations ............................................................... 429

16 Classes Having Pointers To Members ...................................................... 431
   16.1 Pointers to members: an example ......................................................... 431
   16.2 Defining pointers to members .............................................................. 432
   16.3 Using pointers to members .................................................................. 434
   16.4 Pointers to static members .................................................................. 437
   16.5 Pointer sizes ........................................................................................ 438

17 Nested Classes ........................................................................................... 441
   17.1 Defining nested class members ............................................................. 443
   17.2 Declaring nested classes ..................................................................... 444
   17.3 Accessing private members in nested classes ....................................... 444
   17.4 Nesting enumerations ......................................................................... 448
      17.4.1 Empty enumerations ................................................................... 450
17.5 Revisiting virtual constructors .................................................. 451

18 The Standard Template Library .................................................. 453
18.1 Predefined function objects ..................................................... 453
  18.1.1 Arithmetic function objects .............................................. 455
  18.1.2 Relational function objects ............................................. 458
  18.1.3 Logical function objects ............................................... 459
  18.1.4 Function adaptors ....................................................... 460
18.2 Iterators ................................................................................ 463
  18.2.1 std::distance .................................................................. 466
  18.2.2 Insert iterators ............................................................... 467
  18.2.3 Iterators for ‘istream’ objects ......................................... 468
  18.2.4 Iterators for ‘ostream’ objects ......................................... 470
18.3 The class ‘unique_ptr’ .............................................................. 471
  18.3.1 Defining ‘unique_ptr’ objects .......................................... 472
  18.3.2 Creating a plain ‘unique_ptr’ ........................................... 473
  18.3.3 Moving another ‘unique_ptr’ ........................................... 473
  18.3.4 Pointing to a newly allocated object .................................. 474
  18.3.5 Operators and members .................................................. 476
  18.3.6 Using ‘unique_ptr’ objects for arrays ................................. 477
  18.3.7 The legacy class ‘auto_ptr’ (deprecated) ............................. 477
18.4 The class ‘shared_ptr’ ............................................................... 478
  18.4.1 Defining ‘shared_ptr’ objects .......................................... 478
  18.4.2 Creating a plain ‘shared_ptr’ ........................................... 479
  18.4.3 Pointing to a newly allocated object ................................. 479
  18.4.4 Operators and members .................................................. 480
  18.4.5 Casting shared pointers ................................................. 481
  18.4.6 Using ‘shared_ptr’ objects for arrays ............................... 483
18.5 Using ‘make_shared’ to combine ‘shared_ptr’ and ‘new’ .......... 484
18.6 Classes having pointer data members ................................. 484
18.7 Lambda expressions .............................................................. 487
18.8 Regular Expressions ............................................................... 492
18.8.1 The regular expression mini language ............................................. 493
18.8.2 Defining regular expressions: std::regex ........................................ 495
18.8.3 Retrieving matches: std::match_results ........................................ 498
18.8.4 Regular expression matching functions ......................................... 503

18.9 Randomization and Statistical Distributions ...................................... 507
18.9.1 Random Number Generators ...................................................... 508
18.9.2 Statistical distributions ............................................................. 508

19 The STL Generic Algorithms .......................................................... 525
19.1 The Generic Algorithms ............................................................... 525
  19.1.1 accumulate .......................................................... 527
  19.1.2 adjacent_difference ..................................................... 527
  19.1.3 adjacent_find ............................................................. 528
  19.1.4 binary_search ............................................................... 530
  19.1.5 copy .......................................................... 531
  19.1.6 copy_backward ........................................................... 532
  19.1.7 count .......................................................... 532
  19.1.8 count_if ................................................................. 533
  19.1.9 equal ........................................................ 534
  19.1.10 equal_range ............................................................ 535
  19.1.11 fill .......................................................... 537
  19.1.12 fill_n .............................................................. 537
  19.1.13 find .................................................. 538
  19.1.14 find_end ............................................................ 539
  19.1.15 find_first_of .......................................................... 540
  19.1.16 find_if .............................................................. 542
  19.1.17 for_each .............................................................. 543
  19.1.18 generate ............................................................. 545
  19.1.19 generate_n ............................................................ 546
  19.1.20 includes ............................................................. 547
  19.1.21 inner_product .......................................................... 549
  19.1.22 inplace_merge .......................................................... 550
19.1.23 iter_swap ........................................ 551
19.1.24 lexicographical_compare .......................... 552
19.1.25 lower_bound ....................................... 554
19.1.26 max ................................................. 556
19.1.27 max_element ....................................... 556
19.1.28 merge ................................................. 557
19.1.29 min .................................................. 559
19.1.30 min_element ....................................... 560
19.1.31 mismatch ........................................... 561
19.1.32 next_permutation ................................ 562
19.1.33 nth_element ....................................... 564
19.1.34 partial_sort ........................................ 565
19.1.35 partial_sort_copy .................................. 566
19.1.36 partial_sum ........................................ 567
19.1.37 partition ............................................ 568
19.1.38 prev_permutation .................................. 569
19.1.39 random_shuffle ................................... 570
19.1.40 remove .............................................. 572
19.1.41 remove_copy ....................................... 573
19.1.42 remove_copy_if .................................... 574
19.1.43 remove_if ........................................... 575
19.1.44 replace .............................................. 576
19.1.45 replace_copy ....................................... 577
19.1.46 replace_copy_if ................................... 577
19.1.47 replace_if .......................................... 578
19.1.48 reverse ............................................. 579
19.1.49 reverse_copy ...................................... 580
19.1.50 rotate .............................................. 580
19.1.51 rotate_copy ....................................... 581
19.1.52 search .............................................. 582
19.1.53 search_n ........................................... 583
19.1.54 set_difference .................................... 584

xiii
19.1.55 set_intersection .......................................................... 585
19.1.56 set_symmetric_difference ........................................... 587
19.1.57 set_union .............................................................. 588
19.1.58 sort ................................................................. 589
19.1.59 stable_partition ...................................................... 590
19.1.60 stable_sort ........................................................ 591
19.1.61 swap ............................................................... 593
19.1.62 swap_ranges ........................................................ 594
19.1.63 transform .......................................................... 595
19.1.64 unique ............................................................. 597
19.1.65 unique_copy ......................................................... 598
19.1.66 upper_bound ........................................................ 599
19.1.67 Heap algorithms ..................................................... 600
19.2 STL: More function adaptors ............................................. 604
19.2.1 Member function adaptors .......................................... 605
19.2.2 Adaptable functions .................................................. 606

20 Multi Threading .............................................................. 609
20.1 Handling time (absolute and relative) ............................... 610
  20.1.1 Units: the class std::ratio ...................................... 611
  20.1.2 Amounts of time: std::chrono::duration ....................... 612
  20.1.3 Clocks measuring time ........................................... 614
  20.1.4 Points in time: std::chrono::time_point ....................... 615
  20.1.5 Converting time to a NTBS .................................... 616
20.2 Multi Threading .......................................................... 617
  20.2.1 The namespace std::this_thread ................................ 617
  20.2.2 The class std::thread .......................................... 618
20.3 Synchronization (mutexes) .............................................. 628
  20.3.1 Initialization in multi-threaded programs ...................... 630
20.4 Locks and lock handling ................................................. 632
  20.4.1 Deadlocks ........................................................ 636
20.5 Event handling (condition variables) ............................... 638
20.5.1 The class std::condition_variable ........................................... 640
20.5.2 The class std::condition_variable_any ................................. 642
20.5.3 An example using condition variables ................................. 643
20.6 Atomic actions: mutexes not required ......................................... 645
20.7 An example: threaded quicksort .............................................. 650
20.8 Shared States ............................................................................. 653
20.9 Asynchronous return objects: std::future ...................................... 654
   20.9.1 The std::future_error exception and the std::future_errc enum .... 656
20.10 Shared asynchronous return objects: std::shared_future ................. 658
20.11 Starting a new thread: std::async ............................................. 660
20.12 Preparing a task for execution: std::packaged_task ..................... 664
20.13 The class 'std::promise' ................................................................ 668
   20.13.1 Exception propagation: std::exception_ptr ............................ 672
20.14 An example: multi-threaded compilations .................................. 673

21 Function Templates ................................................................. 679
21.1 Defining function templates ...................................................... 679
   21.1.1 Considerations regarding template parameters ..................... 682
   21.1.2 Late-specified return type .................................................. 684
21.2 Passing arguments by reference (reference wrappers) .................... 686
21.3 Using Local and unnamed types as template arguments .................. 687
21.4 Template parameter deduction ................................................. 688
   21.4.1 Lvalue transformations ...................................................... 690
   21.4.2 Qualification transformations ............................................ 691
   21.4.3 Transformation to a base class .......................................... 692
   21.4.4 The template parameter deduction algorithm ...................... 693
   21.4.5 Template type contractions ............................................. 693
21.5 Declaring function templates .................................................... 694
   21.5.1 Instantiation declarations ................................................ 696
21.6 Instantiating function templates ............................................... 696
   21.6.1 Instantiations: no 'code bloat' .......................................... 698
21.7 Using explicit template types ................................................... 699
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.8</td>
<td>Overloading function templates</td>
<td>700</td>
</tr>
<tr>
<td>21.8.1</td>
<td>An example using overloaded function templates</td>
<td>702</td>
</tr>
<tr>
<td>21.8.2</td>
<td>Ambiguities when overloading function templates</td>
<td>702</td>
</tr>
<tr>
<td>21.8.3</td>
<td>Declaring overloaded function templates</td>
<td>704</td>
</tr>
<tr>
<td>21.9</td>
<td>Specializing templates for deviating types</td>
<td>704</td>
</tr>
<tr>
<td>21.9.1</td>
<td>Avoiding too many specializations</td>
<td>706</td>
</tr>
<tr>
<td>21.9.2</td>
<td>Declaring specializations</td>
<td>707</td>
</tr>
<tr>
<td>21.9.3</td>
<td>Complications when using the insertion operator</td>
<td>707</td>
</tr>
<tr>
<td>21.10</td>
<td>Static assertions</td>
<td>708</td>
</tr>
<tr>
<td>21.11</td>
<td>Numeric limits</td>
<td>709</td>
</tr>
<tr>
<td>21.12</td>
<td>Polymorphous wrappers for function objects</td>
<td>712</td>
</tr>
<tr>
<td>21.13</td>
<td>Compiling template definitions and instantiations</td>
<td>713</td>
</tr>
<tr>
<td>21.14</td>
<td>The function selection mechanism</td>
<td>714</td>
</tr>
<tr>
<td>21.15</td>
<td>Determining the template type parameters</td>
<td>715</td>
</tr>
<tr>
<td>21.16</td>
<td>SFINAE: Substitution Failure Is Not An Error</td>
<td>719</td>
</tr>
<tr>
<td>21.17</td>
<td>Summary of the template declaration syntax</td>
<td>720</td>
</tr>
<tr>
<td>22</td>
<td>Class Templates</td>
<td>721</td>
</tr>
<tr>
<td>22.1</td>
<td>Defining class templates</td>
<td>722</td>
</tr>
<tr>
<td>22.1.1</td>
<td>Constructing the circular queue: CirQue</td>
<td>722</td>
</tr>
<tr>
<td>22.1.2</td>
<td>Non-type parameters</td>
<td>724</td>
</tr>
<tr>
<td>22.1.3</td>
<td>Member templates</td>
<td>726</td>
</tr>
<tr>
<td>22.1.4</td>
<td>CirQue's constructors and member functions</td>
<td>728</td>
</tr>
<tr>
<td>22.1.5</td>
<td>Using CirQue objects</td>
<td>733</td>
</tr>
<tr>
<td>22.1.6</td>
<td>Default class template parameters</td>
<td>734</td>
</tr>
<tr>
<td>22.1.7</td>
<td>Declaring class templates</td>
<td>735</td>
</tr>
<tr>
<td>22.1.8</td>
<td>Preventing template instantiations</td>
<td>735</td>
</tr>
<tr>
<td>22.2</td>
<td>Static data members</td>
<td>737</td>
</tr>
<tr>
<td>22.2.1</td>
<td>Extended use of the keyword ‘typename’</td>
<td>738</td>
</tr>
<tr>
<td>22.3</td>
<td>Specializing class templates for deviating types</td>
<td>741</td>
</tr>
<tr>
<td>22.3.1</td>
<td>Example of a class specialization</td>
<td>742</td>
</tr>
<tr>
<td>22.4</td>
<td>Partial specializations</td>
<td>745</td>
</tr>
</tbody>
</table>
22.4.1 Intermezzo: some simple matrix algebraic concepts ........................................ 746
22.4.2 The Matrix class template .............................................................................. 747
22.4.3 The MatrixRow partial specialization ................................................................. 748
22.4.4 The MatrixColumn partial specialization .............................................................. 750
22.4.5 The 1x1 matrix: avoid ambiguity ....................................................................... 750
22.5 Variadic templates ................................................................................................. 752
   22.5.1 Defining and using variadic templates .............................................................. 753
   22.5.2 Perfect forwarding ......................................................................................... 755
   22.5.3 The unpack operator ....................................................................................... 761
   22.5.4 Non-type variadic templates ........................................................................... 762
22.6 Tuples ..................................................................................................................... 763
22.7 Computing the return type of function objects ...................................................... 764
22.8 Instantiating class templates .................................................................................. 767
22.9 Processing class templates and instantiations ......................................................... 768
22.10 Declaring friends ................................................................................................. 769
    22.10.1 Non-templates used as friends in templates .................................................... 770
    22.10.2 Templates instantiated for specific types as friends .......................................... 772
    22.10.3 Unbound templates as friends ...................................................................... 776
    22.10.4 Extended friend declarations ...................................................................... 779
22.11 Class template derivation ..................................................................................... 779
    22.11.1 Deriving ordinary classes from class templates .............................................. 781
    22.11.2 Deriving class templates from class templates ............................................... 782
    22.11.3 Deriving class templates from ordinary classes .............................................. 785
22.12 Static Polymorphism ............................................................................................ 790
    22.12.1 An example of static polymorphism ............................................................... 791
    22.12.2 Converting dynamic polymorphic classes to static polymorphic classes ......... 794
    22.12.3 Using static polymorphism to avoid reimplementations ................................ 798
22.13 Class templates and nesting ............................................................................... 799
22.14 Constructing iterators ......................................................................................... 802
    22.14.1 Implementing a 'RandomAccessIterator' ...................................................... 804
    22.14.2 Implementing a 'reverse_iterator' ................................................................. 809
24 Concrete Examples

24.1 Using file descriptors with 'streambuf' classes 877
   24.1.1 Classes for output operations 877
   24.1.2 Classes for input operations 881
   24.1.3 Fixed-sized field extraction from istream objects 891

24.2 The 'fork' system call 895
   24.2.1 A basic Fork class 896
   24.2.2 Parents and Children 898
   24.2.3 Redirection revisited 899
   24.2.4 The 'Daemon' program 900
   24.2.5 The class 'Pipe' 901
   24.2.6 The class 'ParentSlurp' 903
   24.2.7 Communicating with multiple children 905

24.3 Function objects performing bitwise operations 919

24.4 Adding binary operators to classes 920
   24.4.1 Binary operators allowing promotions 923

24.5 Range-based for-loops and pointer-ranges 925

24.6 Distinguishing lvalues from rvalues with operator[]( ) 927

24.7 Implementing a 'reverse_iterator' 930

24.8 Using 'bison++' and 'flexc++' 933
   24.8.1 Using 'flexc++' to create a scanner 934
   24.8.2 Using 'bison++' and 'flexc++' 938
   24.8.3 Bison++: using polymorphic semantic values 947
Chapter 1

Overview Of The Chapters

The chapters of the C++ Annotations cover the following topics:

- Chapter 1: This overview of the chapters.
- Chapter 2: A general introduction to C++.
- Chapter 3: A first impression: differences between C and C++.
- Chapter 4: Name Spaces: how to avoid name collisions.
- Chapter 5: The ‘string’ data type.
- Chapter 6: The C++ I/O library.
- Chapter 7: The ‘class’ concept: structs having functions. The ‘object’ concept: variables of a class.
- Chapter 8: Static data and functions: members of a class not bound to objects.
- Chapter 9: Allocation and returning unused memory: new, delete, and the function set_new_handler().
- Chapter 10: Exceptions: handle errors where appropriate, rather than where they occur.
- Chapter 11: Give your own meaning to operators.
- Chapter 12: Abstract Containers to put stuff into.
- Chapter 13: Building classes upon classes: setting up class hierarchies.
- Chapter 14: Changing the behavior of member functions accessed through base class pointers.
- Chapter 15: Gaining access to private parts: friend functions and classes.
- Chapter 16: Classes having pointers to members: pointing to locations inside objects.
- Chapter 17: Constructing classes and enums within classes.
- Chapter 18: The Standard Template Library.
- Chapter 19: The STL generic algorithms.
- Chapter 20: Multi Threading.
• Chapter 21: Function templates: using *molds* for type independent functions.
• Chapter 22: Class templates: using *molds* for type independent classes.
• Chapter 23: Advanced Template Use: programming the compiler.
• Chapter 24: Several examples of programs written in C++. 
Chapter 2

Introduction

This document offers an introduction to the C++ programming language. It is a guide for C/C++ programming courses, yearly presented by Frank at the University of Groningen. This document is not a complete C/C++ handbook, as much of the C-background of C++ is not covered. Other sources should be referred to for that (e.g., the Dutch book De programmeertaal C, Brokken and Kubat, University of Groningen, 1996) or the on-line book\(^1\) suggested to me by George Danchev (danchev at spnet dot net).

The reader should be forwarned that extensive knowledge of the C programming language is actually assumed. The C++ Annotations continue where topics of the C programming language end, such as pointers, basic flow control and the construction of functions.

Some elements of the language, like specific lexical tokens (such as trigraphs (such as ??< for {, and using ??> for })), and bigraphs (such as <: for [, and :: for ]) are not covered by the C++ Annotations, as these tokens occur extremely seldom in C++ source code. The reader is referred to the document covering the C++11 standard for an overview.

The version number of the C++ Annotations (currently 10.1.0) is updated when the contents of the document change. The first number is the major number, and is probably not going to change for some time: it indicates a major rewriting. The middle number is increased when new information is added to the document. The last number only indicates small changes; it is increased when, e.g., series of typos are corrected.

This document is published by the Center of Information Technology, University of Groningen, the Netherlands under the GNU General Public License\(^2\).

The C++ Annotations were typeset using the yodl\(^3\) formatting system.

All correspondence concerning suggestions, additions, improvements or changes to this document should be directed to the author:

Frank B. Brokken  
Center of Information Technology,  
University of Groningen  
Nettelbosje 1,

\(^1\)http://publications.gbdirect.co.uk/c_book/  
\(^2\)http://www.gnu.org/licenses/  
\(^3\)http://yodl.sourceforge.net
In this chapter an overview of C++’s defining features is presented. A few extensions to C are reviewed and the concepts of object based and object oriented programming (OOP) are briefly introduced.

2.1 What’s new in the C++ Annotations

This section is modified when the first or second part of the version number changes (and sometimes for the third part as well).

- Version 10.1.0 adds several sections about C++11 regular expression handling, supported since the C++11 standard, but which was overlooked when 10.0.0 was published. Coverage of regular expression handling was moved from the Containers chapter to the chapter covering the Standard Template Library.

- Version 10.0.0 adds a new chapter about multi threading, which is formally supported since the C++11 standard. In addition some minor topics (e.g., std::distance, some new syntax elements) were added to the C++ Annotations.

- In version 9.9.0 sections in chapter 18 about specifying absolute and relative time were rewritten and moved to sections of their own. Sections about condition variables were also rewritten.

- Version 9.8.0 adds new sections in chapter 22 about static polymorphism, removes the concrete/a2x section, now made superfluous by string conversion functions, and removes the C++11 indicators from section headers. The C++ standard has by now been implemented in Gnu’s g++ compiler (4.8.2). Note that the -std=c++11 compiler flag is still necessary if you want the compiler to activate the C++11 extensions.

- Version 9.7.0/9.7.1 adds several new sections (about time specifications, system_error, error_category, error_code, this_thread, locks) and other sections received a major overhaul (multi-threading, lambda expressions, the main function).

- Version 9.6.0 adds a section about noexcept, deprecating throw lists. Also the string chapter was updated.

- Version 9.5.0 adapts the abstract containers to C++11, and adds a new section (Allocators) just before introducing the sequential containers. make_shared, combining shared_ptr and (new).

- Version 9.4.0 adds a new section to chapter 18 about make_shared, combining shared_ptr and (new).

- Version 9.3.0 refines the coverage of the static_cast and reinterpret_cast, following a suggestion provided by Gido Schoenmacker.

- There are two major differences between versions 9.2.0 and 9.1.0. First, unrestricted unions are covered in more detail (cf. section 12.6). Second, by now flexcpp\(^4\) has been released, and the sections previously using flex (cf. section 24.8) are now using flexc++.

\(^4\)http://flexcpp.org/
2.1. WHAT'S NEW IN THE C++ ANNOTATIONS

- Version 9.1.0 adds several new sections describing elements of the language that by now have been implemented in Gnu's g++ compiler version 4.7. In the Annotations's contents these sections are clearly marked as C++11, 4.7. For section marked by merely C++11 it is assumed that at least Gnu's compiler version 4.6 is available. Sections marked as C++, ? refer to elements in the C++11 (C++11) standard that haven't been implemented yet in Gnu's g++ compiler. Since C++11 is now the 'official' name of the new standard, references to C++0x have been replaced by C++11.

Installation limits of various integral types are frequently obtained using #defines set in the <climits> header file. However, the numeric_limits template offers a (preferred) alternative, as numeric_limits can also be used when defining templates. See chapter 21 for details.

To the distribution's ./contributions directory I added Jurjen Bokma's (jurjen dot bokma at rug dot nl) 'makebook' recipe for creating a neatly bound C++ Annotations book. The result is fabulous! Thanks, Jurjen!

From now on, this 'what's new' overview of changes to the Annotations is restricted to the current and previous major release. Previous modifications can be found in the distribution's whatsnew.yo.old file.

Finally, typos were repaired.

- Version 9.0.0 was released following an extensive discussion with several members of the C++ standards committee about the form of move special members (move constructors, move assignment operators, other functions defining rvalue type parameters). This discussion, in particular the discussions I had with Dave Abrahams, Jonathan Wakely and Herb Sutter resulted, eventually, in the C++ Annotations relaxing the principle of const-correctness, and in modifying the declarations and implementations of move special members in this release. This shift in position (adopted by the C++ Annotations since its very early releases) profoundly affects much of the C++ Annotations's contents, and warrants an upgrade to the next major release.

The principle of const-correctness has always been visible in the C++ Annotations, defining return values of arithmetic binary operators like operator+ as const return values. Here the C++ Annotations also applied another principle: 'do as the ints do'. When (e.g.,) adding two int-values the result is customarily considered immutable. I.e., (a + 5) += 4 makes no sense, with the compiler refusing to compile the statement. But is a + 5 a constant? There is no simple answer to that question. Before the advent of C++11 I thought the answer was 'yes', but strangely enough, the answer was not always 'yes'. If the above a is of type std::string then (a + "b") += "c" suddenly is accepted by the compiler. The C++ Annotations never adopted this scheme, but stuck to the rule 'do as the ints do' by defining the return types of functions returning values as const values.

C++11 added rvalue references to the language, and then I eventually was convinced that defining const return values should in general be avoided. As C++11 allows temporaries to be associated with rvalue references, a completely new situation is created. Suddenly intermediate int values can be modified, as illustrated by the following snippet of code:

```cpp
void fun(int &&tmp)
{
    tmp += 4; // compiles OK
}
int main()
{
    int a = 8;
    fun(a + 5);
}
```
The snippet of code also shows the standard definition of an rvalue reference as an entity of type `Type &&`. This definition of rvalue reference parameters is now used all over the C++ Annotations, together with using non-const return types of functions returning values.

Many readers have submitted suggestions for improvements since version 8.3.1 was released. A big ‘thank you’ to all of you, but in particular to Francesco Poli who continued to send in suggestions for improvements for a period of almost two years. His suggestions were an invaluable source of improvement for almost every single section of the C++ Annotations. Thanks, Francesco, for all the effort you’ve put in improving this document!

Finally, in version 9.0.0 sections were added and sometimes moved. The section about unrestricted unions was completed and moved to the ‘Containers’ chapter, and a new section about adding binary operators to classes using function templates was added to the C++ Annotations’ final chapter (concrete examples).

• Version 8.3.1: usually a subminor version isn’t explicitly mentioned in this section, but in this case the changes from 8.3.0 to 8.3.1 were the result of many, many small and not so small corrections submitted by Francesco Poli who did a very thorough close reading job on the Annotations. Thanks again, Francesco, for all your contributions!

• Version 8.3.0 adds sections about various (member) function adaptors and adds/rephrases several sections about statistical distribution functions (chapter 18). When covering elements from the C++-0x standard it is assumed that the Gnu g++ compiler version 4.4 is available. With elements of the C++-0x standard requiring versions beyond 4.4 the required versions are explicitly mentioned, if already known. All suggestions sent in by various readers have also been processed, their help to improve the quality of the C++ Annotations is greatly appreciated: thanks!

• Version 8.2.0 adds a section about casting shared_ptrs (section 18.4.5) and about sharing arrays of objects (18.4.6).

• Version 8.1.0 was released following a complete overhaul of the C++ Annotations, with two pre-leases in between. Many inconsistencies that had crept into the text and examples were removed, streamlining the text and synchronizing examples with the text. All of the code examples have received a work-over, replacing endl by ‘\n’, making virtual functions private, etc., etc. The sections labeled C++11 were improved and sections showing C++11 now also mention the g++ version in which the new feature will be made available, using if this is as yet unknown. No version is shown if the feature is already available in g++ 4.3 (or in one of its subreleases, like 4.3.3). I received a host of suggestions from Francesco Poli (thanks, Francesco (and several others), for all the effort you’ve put into sending me those corrections).

• Version 8.0.0 was released as a result of the upcoming new C++ standard becoming (partially) available in the Gnu g++ compiler. Not all new elements of the new standard (informally called the C++0x standard) are available right now, and new subreleases of the C++ Annotations will appear once more elements become implemented in the g++ compiler. In section 2.2.3 the way to activate the new standard is shown, and new sections covering elements of the new standard show C++11 in their section-titles.

Furthermore, two new chapters were added: the STL chapter is now split in two. The STL chapter now covers the STL except for the Generic Algorithms which are now discussed in a separate chapter. Name spaces, originally covered by the introductory chapter are now also covered in a separate chapter.

5http://en.wikipedia.org/wiki/C++11
6http://gcc.gnu.org/projects/cxx0x.html
2.2 C++’s history

The first implementation of C++ was developed in the 1980s at the AT&T Bell Labs, where the Unix operating system was created.

C++ was originally a ‘pre-compiler’, similar to the preprocessor of C, converting special constructions in its source code to plain C. Back then this code was compiled by a standard C compiler. The ‘pre-code’, which was read by the C++ pre-compiler, was usually located in a file with the extension .cc, .C or .cpp. This file would then be converted to a C source file with the extension .c, which was thereupon compiled and linked.

The nomenclature of C++ source files remains: the extensions .cc and .cpp are still used. However, the preliminary work of a C++ pre-compiler is nowadays usually performed during the actual compilation process. Often compilers determine the language used in a source file from its extension. This holds true for Borland’s and Microsoft’s C++ compilers, which assume a C++ source for an extension .cpp. The Gnu compiler g++, which is available on many Unix platforms, assumes for C++ the extension .cc.

The fact that C++ used to be compiled into C code is also visible from the fact that C++ is a superset of C: C++ offers the full C grammar and supports all C-library functions, and adds to these features of its own. This makes the transition from C to C++ quite easy. Programmers familiar with C may start ‘programming in C++’ by using source files having extensions .cc or .cpp instead of .c, and may then comfortably slip into all the possibilities offered by C++. No abrupt change of habits is required.

2.2.1 History of the C++ Annotations

The original version of the C++ Annotations was written by Frank Brokken and Karel Kubat in Dutch using LaTeX. After some time, Karel rewrote the text and converted the guide to a more suitable format and (of course) to English in september 1994.

The first version of the guide appeared on the net in october 1994. By then it was converted to SGML.

Gradually new chapters were added, and the contents were modified and further improved (thanks to countless readers who sent us their comment).

In major version four Frank added new chapters and converted the document from SGML to yodl.\footnote{http://yodl.sourceforge.net}

The C++ Annotations are freely distributable. Be sure to read the legal notes.\footnote{legal.shtml}

Reading the annotations beyond this point implies that you are aware of these notes and that you agree with them.

If you like this document, tell your friends about it. Even better, let us know by sending email to Frank.\footnote{mailto:f.h.brokken@rug.nl}

2.2.2 Compiling a C program using a C++ compiler

Prospective C++ programmers should realize that C++ is not a perfect superset of C. There are some differences you might encounter when you simply rename a file to a file having the extension .cc.
and run it through a C++ compiler:

- In C, \texttt{sizeof('c')} equals \texttt{sizeof(int)}, 'c' being any ASCII character. The underlying philosophy is probably that \texttt{char}s, when passed as arguments to functions, are passed as integers anyway. Furthermore, the C compiler handles a character constant like 'c' as an integer constant. Hence, in C, the function calls

\begin{verbatim}
    putchar(10);
\end{verbatim}

and

\begin{verbatim}
    putchar('\n');
\end{verbatim}

are synonymous.

By contrast, in C++, \texttt{sizeof('c')} is always 1 (but see also section 3.4.2). An \texttt{int} is still an \texttt{int}, though. As we shall see later (section 2.5.4), the two function calls

\begin{verbatim}
    somefunc(10);
\end{verbatim}

and

\begin{verbatim}
    somefunc('\n');
\end{verbatim}

may be handled by different functions: C++ distinguishes functions not only by their names, but also by their argument types, which are different in these two calls. The former using an \texttt{int} argument, the latter a \texttt{char}.

- C++ requires very strict prototyping of external functions. E.g., in C a prototype like

\begin{verbatim}
    void func();
\end{verbatim}

means that a function \texttt{func()} exists, returning no value. The declaration doesn’t specify which arguments (if any) are accepted by the function.

However, in C++ the above declaration means that the function \texttt{func()} does \textit{not} accept any arguments at all. Any arguments passed to it result in a compile-time error.

Note that the keyword \texttt{extern} is not required when declaring functions. A function definition becomes a function declaration simply by replacing a function’s body by a semicolon. The keyword \texttt{extern} is required, though, when declaring variables.

### 2.2.3 Compiling a C++ program

To compile a C++ program, a C++ compiler is required. Considering the free nature of this document, it won’t come as a surprise that a free compiler is suggested here. The Free Software Foundation (FSF) provides at http://www.gnu.org a free C++ compiler which is, among other places, also part of the Debian (http://www.debian.org) distribution of Linux (http://www.debian.org).

To use the features of the C++11 standard the compiler flag \texttt{-std=c++11} must currently be provided. In the C++ Annotations it is assumed that this flag is used when compiling the examples.
2.2. C++’S HISTORY

2.2.3.1 C++ under MS-Windows

For MS-Windows Cygnus (http://sources.redhat.com/cygwin) provides the foundation for installing the Windows port of the Gnu g++ compiler.

When visiting the above URL to obtain a free g++ compiler, click on install now. This will download the file setup.exe, which can be run to install cygwin. The software to be installed can be downloaded by setup.exe from the internet. There are alternatives (e.g., using a CD-ROM), which are described on the Cygwin page. Installation proceeds interactively. The offered defaults are sensible and should be accepted unless you have reasons to divert.

The most recent Gnu g++ compiler can be obtained from http://gcc.gnu.org. If the compiler that is made available in the Cygnus distribution lags behind the latest version, the sources of the latest version can be downloaded after which the compiler can be built using an already available compiler. The compiler’s webpage (mentioned above) contains detailed instructions on how to proceed. In our experience building a new compiler within the Cygnus environment works flawlessly.

2.2.3.2 Compiling a C++ source text

Generally the following command can be used to compile a C++ source file ‘source.cc’:

```g++ source.cc```

This produces a binary program (a.out or a.exe). If the default name is inappropriate, the name of the executable can be specified using the –o flag (here producing the program source):

```g++ -o source source.cc```

If a mere compilation is required, the compiled module can be produced using the –c flag:

```g++ -c source.cc```

This generates the file source.o, which can later on be linked to other modules. As pointed out, provide the compiler option –std=c++11 (note: two dashes) to activate the features of the C++11 standard.

C++ programs quickly become too complex to maintain ‘by hand’. With all serious programming projects program maintenance tools are used. Usually the standard make program is used to maintain C++ programs, but good alternatives exist, like the icmake or ccbuild program maintenance utilities.

It is strongly advised to start using maintenance utilities early in the study of C++.

2.2.3.3 C++14

On August 18, 2014 the ISO/IEC standard committee on C++ approved an update of the C++11 standard, called c14(), offering some improvements over C++11, and some bugfixes. The C++14 working draft of the standard is freely available, and can be downloaded from here. To activate the C++14 standard in Gnu’s g++ compiler, specify std=c++14.

10http://icmake.sourceforge.net/
11http://ccbuild.sourceforge.net/
12http://cppannotations.sourceforge.net/n3797.pdf
In time the C++ Annotations will reflect the changes related to the C++14 standard. Refer to the C++14 entry in the index for locations where the C++14 standard is mentioned.

2.3 C++: advantages and claims

Often it is said that programming in C++ leads to ‘better’ programs. Some of the claimed advantages of C++ are:

- New programs would be developed in less time because old code can be reused.
- Creating and using new data types would be easier than in C.
- The memory management under C++ would be easier and more transparent.
- Programs would be less bug-prone, as C++ uses a stricter syntax and type checking.
- ‘Data hiding’, the usage of data by one program part while other program parts cannot access the data, would be easier to implement with C++.

Which of these allegations are true? Originally, our impression was that the C++ language was somewhat overrated; the same holding true for the entire object-oriented programming (OOP) approach. The enthusiasm for the C++ language resembles the once uttered allegations about Artificial-Intelligence (AI) languages like Lisp and Prolog: these languages were supposed to solve the most difficult AI-problems ‘almost without effort’. New languages are often oversold: in the end, each problem can be coded in any programming language (say BASIC or assembly language). The advantages and disadvantages of a given programming language aren’t in ‘what you can do with them’, but rather in ‘which tools the language offers to implement an efficient and understandable solution to a programming problem’. Often these tools take the form of syntactic restrictions, enforcing or promoting certain constructions or which simply suggest intentions by applying or ‘embracing’ such syntactic forms. Rather than a long list of plain assembly instructions we now use flow control statements, functions, objects or even (with C++) so-called templates to structure and organize code and to express oneself ‘eloquently’ in the language of one’s choice.

Concerning the above allegations of C++, we support the following, however:

- The development of new programs while existing code is reused can also be implemented in C by, e.g., using function libraries. Functions can be collected in a library and need not be re-invented with each new program. C++, however, offers specific syntax possibilities for code reuse, apart from function libraries (see chapters 13 and 21).
- Creating and using new data types is certainly possible in C; e.g., by using structs, typedefs etc. From these types other types can be derived, thus leading to structs containing structs and so on. In C++ these facilities are augmented by defining data types which are completely ‘self supporting’, taking care of, e.g., their memory management automatically (without having to resort to an independently operating memory management system as used in, e.g., Java).
- In C++ memory management can in principle be either as easy or as difficult as it is in C. Especially when dedicated C functions such as xmalloc and xrealloc are used (allocating the memory or aborting the program when the memory pool is exhausted). However, with functions like malloc it is easy to err. Frequently errors in C programs can be traced back to miscalculations when using malloc. Instead, C++ offers facilities to allocate memory in a somewhat safer way, using its operator new.
2.4 WHAT IS OBJECT-ORIENTED PROGRAMMING?

- Concerning 'bug proneness' we can say that C++ indeed uses stricter type checking than C. However, most modern C compilers implement 'warning levels'; it is then the programmer's choice to disregard or get rid of the warnings. In C++ many of such warnings become fatal errors (the compilation stops).

- As far as 'data hiding' is concerned, C does offer some tools. E.g., where possible, local or static variables can be used and special data types such as structs can be manipulated by dedicated functions. Using such techniques, data hiding can be implemented even in C; though it must be admitted that C++ offers special syntactic constructions, making it far easier to implement 'data hiding' (and more in general: 'encapsulation') in C++ than in C.

C++ in particular (and OOP in general) is of course not the solution to all programming problems. However, the language does offer various new and elegant facilities which are worth investigating. At the downside, the level of grammatical complexity of C++ has increased significantly as compared to C. This may be considered a serious drawback of the language. Although we got used to this increased level of complexity over time, the transition was neither fast nor painless.

With the C++ Annotations we hope to help the reader when transiting from C to C++ by focusing on the additions of C++ as compared to C and by leaving out plain C. It is our hope that you like this document and may benefit from it.

Enjoy and good luck on your journey into C++!

2.4 What is Object-Oriented Programming?

Object-oriented (and object-based) programming propagates a slightly different approach to programming problems than the strategy usually used in C programs. In C programming problems are usually solved using a 'procedural approach': a problem is decomposed into subproblems and this process is repeated until the subtasks can be coded. Thus a conglomerate of functions is created, communicating through arguments and variables, global or local (or static).

In contrast (or maybe better: in addition) to this, an object-based approach identifies the keywords used in a problem statement. These keywords are then depicted in a diagram where arrows are drawn between those keywords to depict an internal hierarchy. The keywords become the objects in the implementation and the hierarchy defines the relationship between these objects. The term object is used here to describe a limited, well-defined structure, containing all information about an entity: data types and functions to manipulate the data. As an example of an object oriented approach, an illustration follows:

The employees and owner of a car dealer and auto garage company are paid as follows. First, mechanics who work in the garage are paid a certain sum each month. Second, the owner of the company receives a fixed amount each month. Third, there are car salesmen who work in the showroom and receive their salary each month plus a bonus per sold car. Finally, the company employs second-hand car purchasers who travel around; these employees receive their monthly salary, a bonus per bought car, and a restitution of their travel expenses.

When representing the above salary administration, the keywords could be mechanics, owner, salesmen and purchasers. The properties of such units are: a monthly salary, sometimes a bonus per purchase or sale, and sometimes restitution of travel expenses. When analyzing the problem in this manner we arrive at the following representation:

- The owner and the mechanics can be represented by identical types, receiving a given salary
per month. The relevant information for such a type would be the monthly amount. In addition this object could contain data as the name, address and social security number.

- Car salesmen who work in the showroom can be represented as the same type as above but with some extra functionality: the number of transactions (sales) and the bonus per transaction.
  
  In the hierarchy of objects we would define the dependency between the first two objects by letting the car salesmen be ‘derived’ from the owner and mechanics.

- Finally, there are the second-hand car purchasers. These share the functionality of the salesmen except for travel expenses. The additional functionality would therefore consist of the expenses made and this type would be derived from the salesmen.

The hierarchy of the identified objects are further illustrated in Figure 2.1.

The overall process in the definition of a hierarchy such as the above starts with the description of the most simple type. Traditionally (and still in vogue with some popular object oriented languages) more complex types are then derived from the basic set, with each derivation adding a little extra functionality. From these derived types, more complex types can be derived ad infinitum, until a representation of the entire problem can be made. Over the years, however, this approach has become less popular in C++ as it typically results in overly tight coupling, which in turns reduces rather than enhances the understanding, maintainability and testability of complex programs. In C++ object oriented program more and more favors small, easy to understand hierarchies, limited coupling and a developmental process where design patterns (cf. Gamma et al. (1995)) play a central role.

Nonetheless, in C++ classes are frequently used to define the characteristics of objects. Classes contain the necessary functionality to do useful things. Classes generally do not offer all their functionality (and typically none of their data) to objects of other classes. As we will see, classes tend to hide their properties in such a way that they are not directly modifiable by the outside world. Instead, dedicated functions are used to reach or modify the properties of objects. Thus class-type objects are able to uphold their own integrity. The core concept here is encapsulation of which data hiding is just an example. These concepts are further explained in chapter 7.

2.5 Differences between C and C++

In this section some examples of C++ code are shown. Some differences between C and C++ are highlighted.
2.5. DIFFERENCES BETWEEN C AND C++

2.5.1 The function ‘main’

In C++ there are only two variants of the function `main`: `int main()` and `int main(int argc, char **argv)`.

Notes:

- The return type of `main` is `int`, and not `void`;
- The function `main` cannot be overloaded (for other than the abovementioned signatures);
- It is not required to use an explicit return statement at the end of `main`. If omitted `main` returns 0;
- The value of `argv[argc]` equals 0;
- The ‘third char **envp parameter’ is not defined by the C++ standard and should be avoided. Instead, the global variable `extern char **environ` should be declared providing access to the program’s environment variables. Its final element has the value 0;
- A C++ program ends normally when the `main` function returns. Using a function try block (cf. section 10.11) for `main` is also considered a normal end of a C++ program. When a C++ ends normally, destructors (cf. section 9.2) of globally defined objects are activated. A function like `exit(3)` does not normally end a C++ program and using such functions is therefore deprecated.

2.5.2 End-of-line comment

According to the ANSI/ISO definition, ‘end of line comment’ is implemented in the syntax of C++. This comment starts with `//` and ends at the end-of-line marker. The standard C comment, delimited by `/*` and `*/` can still be used in C++:

```c
int main()
{
    // this is end-of-line comment
    // one comment per line

    /*
    this is standard-C comment, covering
    multiple lines
    */

    return 0;
}
```

Despite the example, it is advised not to use C type comment inside the body of C++ functions. Sometimes existing code must temporarily be suppressed, e.g., for testing purposes. In those cases it’s very practical to be able to use standard C comment. If such suppressed code itself contains such comment, it would result in nested comment-lines, resulting in compiler errors. Therefore, the rule of thumb is not to use C type comment inside the body of C++ functions (alternatively, `#if 0` until `#endif` pair of preprocessor directives could of course also be used).

2.5.3 Strict type checking

C++ uses very strict type checking. A prototype must be known for each function before it is called, and the call must match the prototype. The program
```
int main()
{
    printf("Hello World\n");
}
```

often compiles under C, albeit with a warning that `printf()` is an unknown function. But C++ compilers (should) fail to produce code in such cases. The error is of course caused by the missing `#include <stdio.h>` (which in C++ is more commonly included as `#include <cstdio>` directive).

And while we’re at it: as we’ve seen in C++ `main` always uses the `int` return value. Although it is possible to define `int main()` without explicitly defining a return statement, within `main` it is not possible to use a `return` statement without an explicit `int`-expression. For example:

```
int main()
{
    return;  // won’t compile: expects int expression, e.g.
    // return 1;
}
```

### 2.5.4 Function Overloading

In C++ it is possible to define functions having identical names but performing different actions. The functions must differ in their parameter lists (and/or in their `const` attribute). An example is given below:

```
#include <stdio.h>

void show(int val)
{
    printf("Integer: %d\n", val);
}

void show(double val)
{
    printf("Double: %lf\n", val);
}

void show(char const *val)
{
    printf("String: %s\n", val);
}

int main()
{
    show(12);
    show(3.1415);
    show("Hello World!\n");
}
```

In the above program three functions `show` are defined, only differing in their parameter lists, expecting an `int`, `double` and `char *`, respectively. The functions have identical names. Functions
having identical names but different parameter lists are called *overloaded*. The act of defining such functions is called ‘function overloading’.

The C++ compiler implements function overloading in a rather simple way. Although the functions share their names (in this example show), the compiler (and hence the linker) use quite different names. The conversion of a name in the source file to an internally used name is called ‘name mangling’. E.g., the C++ compiler might convert the prototype `void show(int)` to the internal name `VshowI`, while an analogous function having a `char *` argument might be called `VshowCP`. The actual names that are used internally depend on the compiler and are not relevant for the programmer, except where these names show up in e.g., a listing of the contents of a library.

Some additional remarks with respect to function overloading:

- Do not use function overloading for functions doing conceptually different tasks. In the example above, the functions `show` are still somewhat related (they print information to the screen). However, it is also quite possible to define two functions `lookup`, one of which would find a name in a list while the other would determine the video mode. In this case the behavior of those two functions have nothing in common. It would therefore be more practical to use names which suggest their actions; say, `findname` and `videoMode`.

- C++ does not allow identically named functions to differ only in their return values, as it is always the programmer’s choice to either use or ignore a function’s return value. E.g., the fragment

```c
printf("Hello World\n");
```

provides no information about the return value of the function `printf`. Two functions `printf` which only differ in their return types would therefore not be distinguishable to the compiler.

- In chapter 7 the notion of `const` member functions is introduced (cf. section 7.7). Here it is merely mentioned that classes normally have so-called member functions associated with them (see, e.g., chapter 5 for an informal introduction to the concept). Apart from overloading member functions using different parameter lists, it is then also possible to overload member functions by their `const` attributes. In those cases, classes may have pairs of identically named member functions, having identical parameter lists. Then, these functions are overloaded by their `const` attribute. In such cases only one of these function must have the `const` attribute.

### 2.5.5 Default function arguments

In C++ it is possible to provide ‘default arguments’ when defining a function. These arguments are supplied by the compiler when they are not specified by the programmer. For example:

```c
#include <stdio.h>

void showstring(char *str = "Hello World!\n");

int main()
{
    showstring("Here’s an explicit argument.\n");

    showstring(); // in fact this says:
    // showstring("Hello World!\n");
}
```
The possibility to omit arguments in situations where default arguments are defined is just a nice touch: it is the compiler who supplies the lacking argument unless it is explicitly specified at the call. The code of the program will neither be shorter nor more efficient when default arguments are used.

Functions may be defined with more than one default argument:

```c
void two_ints(int a = 1, int b = 4);

int main()
{
    two_ints(); // arguments: 1, 4
    two_ints(20); // arguments: 20, 4
    two_ints(20, 5); // arguments: 20, 5
}
```

When the function `two_ints` is called, the compiler supplies one or two arguments whenever necessary. A statement like `two_ints(,6)` is, however, not allowed: when arguments are omitted they must be on the right-hand side.

Default arguments must be known at compile-time since at that moment arguments are supplied to functions. Therefore, the default arguments must be mentioned at the function’s declaration, rather than at its implementation:

```c
#include <stdio.h>

void show(int val)
{
#include <stdio.h>

void show(int val)
```
2.5. DIFFERENCES BETWEEN C AND C++

```c
{  printf("Integer: %d\n", val);
}

void show(double val)
{
  printf("Double: %lf\n", val);
}

void show(char const *val)
{
  printf("String: %s\n", val);
}

int main()
{
  show(12);
  show(3.1415);
  show("Hello World!\n");
}
```

In this situation a programmer intending to call `show(char const *)` might call `show(0)`. But this doesn’t work, as 0 is interpreted as `int` and so `show(int)` is called. But calling `show(NULL)` doesn’t work either, as C++ usually defines `NULL` as 0, rather than `((void *)0)`. So, `show(int)` is called once again. To solve these kinds of problems the new C++ standard introduces the keyword `nullptr` representing the 0 pointer. In the current example the programmer should call `show(nullptr)` to avoid the selection of the wrong function. The `nullptr` value can also be used to initialize pointer variables. E.g.,

```c
int *ip = nullptr; // OK
int value = nullptr; // error: value is no pointer
```

### 2.5.7 The ‘void’ parameter list

In C, a function prototype with an empty parameter list, such as

```
void func();
```

means that the argument list of the declared function is not prototyped: the compiler does warn against calling `func` with any set of arguments. In C the keyword `void` is used when it is the explicit intent to declare a function with no arguments at all, as in:

```
void func(void);
```

As C++ enforces strict type checking, in C++ an empty parameter list indicates the total absence of parameters. The keyword `void` is thus omitted.

### 2.5.8 The ‘#define __cplusplus’

Each C++ compiler which conforms to the ANSI/ISO standard defines the symbol `__cplusplus`: it is as if each source file were prefixed with the preprocessor directive `#define __cplusplus`.

CHAPTER 2. INTRODUCTION

We shall see examples of the usage of this symbol in the following sections.

### 2.5.9 Using standard C functions

Normal C functions, e.g., which are compiled and collected in a run-time library, can also be used in C++ programs. Such functions, however, must be declared as C functions.

As an example, the following code fragment declares a function `xmalloc` as a C function:

```c
extern "C" void *xmalloc(int size);
```

This declaration is analogous to a declaration in C, except that the prototype is prefixed with `extern "C"`.

A slightly different way to declare C functions is the following:

```c
extern "C"
{
    // C-declarations go in here
}
```

It is also possible to place preprocessor directives at the location of the declarations. E.g., a C header file `myheader.h` which declares C functions can be included in a C++ source file as follows:

```c
extern "C"
{
    #include <myheader.h>
}
```

Although these two approaches may be used, they are actually seldom encountered in C++ sources. A more frequently used method to declare external C functions is encountered in the next section.

### 2.5.10 Header files for both C and C++

The combination of the predefined symbol `__cplusplus` and the possibility to define `extern "C"` functions offers the ability to create header files for both C and C++. Such a header file might, e.g., declare a group of functions which are to be used in both C and C++ programs.

The setup of such a header file is as follows:

```c
#ifdef __cplusplus
extern "C"
{
    #ifdef __cplusplus
    #endif
    /* declaration of C-data and functions are inserted here. E.g., */
    void *xmalloc(int size);
    #ifdef __cplusplus
    
    #endif
#endif
```
Using this setup, a normal C header file is enclosed by `extern "C" {` which occurs near the top of the file and by `}`, which occurs near the bottom of the file. The `#ifdef` directives test for the type of the compilation: C or C++. The ‘standard’ C header files, such as `stdio.h`, are built in this manner and are therefore usable for both C and C++.

In addition C++ headers should support `include guards`. In C++ it is usually undesirable to include the same header file twice in the same source file. Such multiple inclusions can easily be avoided by including an `#ifndef` directive in the header file. For example:

```c
#ifndef MYHEADER_H_
#define MYHEADER_H_
    // declarations of the header file is inserted here,
    // using #ifdef __cplusplus etc. directives
#endif
```

When this file is initially scanned by the preprocessor, the symbol `MYHEADER_H_` is not yet defined. The `#ifndef` condition succeeds and all declarations are scanned. In addition, the symbol `MYHEADER_H_` is defined.

When this file is scanned next while compiling the same source file, the symbol `MYHEADER_H_` has been defined and consequently all information between the `#ifndef` and `#endif` directives is skipped by the compiler.

In this context the symbol name `MYHEADER_H_` serves only for recognition purposes. E.g., the name of the header file can be used for this purpose, in capitals, with an underscore character instead of a dot.

Apart from all this, the custom has evolved to give C header files the extension `.h`, and to give C++ header files no extension. For example, the standard `iostreams` `cin`, `cout` and `cerr` are available after including the header file `iostream` rather than `iostream.h`. In the Annotations this convention is used with the standard C++ header files, but not necessarily everywhere else.

There is more to be said about header files. Section 7.12 provides an in-depth discussion of the preferred organization of C++ header files.

### 2.5.11 Defining local variables

In C local variables can only be defined at the top of a function or at the beginning of a nested block. In C++ local variables can be created at any position in the code, even between statements.

Furthermore, local variables can be defined within some statements, just prior to their usage. A typical example is the `for` statement:

```c
#include <stdio.h>

int main()
{
    for (int i = 0; i < 20; ++i)
        printf("%d\n", i);
}
```

In this program the variable `i` is created in the initialization section of the `for` statement. According to the ANSI-standard, the variable does not exist prior to the `for`-statement and not beyond the
for-statement. With some older compilers, the variable continues to exist after the execution of the for-statement, but nowadays a warning like

    warning: name lookup of ‘i’ changed for new ANSI ‘for’ scoping using obsolete binding at ‘i’

is issued when the variable is used outside of the for-loop.

The implication seems clear: define a variable just before the for-statement if it is to be used beyond that statement. Otherwise the variable should be defined inside the for-statement itself. This reduces its scope as much as possible, which is a very desirable characteristic.

Defining local variables when they’re needed requires a little getting used to. However, eventually it tends to produce more readable, maintainable and often more efficient code than defining variables at the beginning of compound statements. We suggest the following rules of thumb for defining local variables:

- Local variables should be created at ‘intuitively right’ places, such as in the example above. This does not only entail the for-statement, but also all situations where a variable is only needed, say, half-way through the function.
- More in general, variables should be defined in such a way that their scope is as limited and localized as possible. When avoidable local variables are not defined at the beginning of functions but rather where they’re first used.
- It is considered good practice to avoid global variables. It is fairly easy to lose track of which global variable is used for what purpose. In C++ global variables are seldom required, and by localizing variables the well known phenomenon of using the same variable for multiple purposes, thereby invalidating each individual purpose of the variable, can easily be prevented.

If considered appropriate, nested blocks can be used to localize auxiliary variables. However, situations exist where local variables are considered appropriate inside nested statements. The just mentioned for statement is of course a case in point, but local variables can also be defined within the condition clauses of if-else statements, within selection clauses of switch statements and condition clauses of while statements. Variables thus defined are available to the full statement, including its nested statements. For example, consider the following switch statement:

```c
#include <stdio.h>

int main()
{
    switch (int c = getchar())
    {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            printf("Saw vowel %c\n", c);
            break;

        case EOF:
            printf("Saw EOF\n");
            break;
    }
}
```
2.5. DIFFERENCES BETWEEN C AND C++

```c

    case '0' ... '9':
        printf("Saw number character %c\n", c);
    break;

    default:
        printf("Saw other character, hex value 0x%2x\n", c);
    }
}
```

Note the location of the definition of the character ‘c’: it is defined in the expression part of the `switch` statement. This implies that ‘c’ is available only to the `switch` statement itself, including its nested (sub)statements, but not outside the scope of the `switch`.

The same approach can be used with `if` and `while` statements: a variable that is defined in the condition part of an `if` and `while` statement is available in their nested statements. There are some caveats, though:

- The variable definition must result in a variable which is initialized to a numeric or logical value;
- The variable definition cannot be nested (e.g., using parentheses) within a more complex expression.

The latter point of attention should come as no big surprise: in order to be able to evaluate the logical condition of an `if` or `while` statement, the value of the variable must be interpretable as either zero (false) or non-zero (true). Usually this is no problem, but in C++ objects (like objects of the type `std::string` (cf. chapter 5)) are often returned by functions. Such objects may or may not be interpretable as numeric values. If not (as is the case with `std::string` objects), then such variables can not be defined at the condition or expression clauses of condition- or repetition statements. The following example will therefore not compile:

```c
if (std::string myString = getString()) // assume getString returns
    // a std::string value
    // process myString
```

The above example requires additional clarification. Often a variable can profitably be given local scope, but an extra check is required immediately following its initialization. The initialization and the test cannot both be combined in one expression. Instead two nested statements are required. Consequently, the following example won’t compile either:

```c
if ((int c = getchar()) && strchr("aeiou", c))
    printf("Saw a vowel\n");
```

If such a situation occurs, either use two nested `if` statements, or localize the definition of `int c` using a nested compound statement:

```c
if (int c = getchar()) // nested if-statements
    if (strchr("aeiou", c))
        printf("Saw a vowel\n");
{
    // nested compound statement
```
int c = getchar();
if (c && strchr("aeiou", c))
  printf("Saw a vowel\n");
}

2.5.12 The keyword ‘typedef’

The keyword typedef is still used in C++, but is not required anymore when defining union, struct or enum definitions. This is illustrated in the following example:

```cpp
struct SomeStruct
{
  int a;
  double d;
  char string[80];
};
```

When a struct, union or other compound type is defined, the tag of this type can be used as type name (this is SomeStruct in the above example):

```cpp
SomeStruct what;
what.d = 3.1415;
```

2.5.13 Functions as part of a struct

In C++ we may define functions as members of structs. Here we encounter the first concrete example of an object: as previously described (see section 2.4), an object is a structure containing data while specialized functions exist to manipulate those data.

A definition of a struct Point is provided by the code fragment below. In this structure, two int data fields and one function draw are declared.

```cpp
struct Point // definition of a screen-dot
{
  int x; // coordinates
  int y; // x/y
  void draw(); // drawing function
};
```

A similar structure could be part of a painting program and could, e.g., represent a pixel. With respect to this struct it should be noted that:

- The function draw mentioned in the struct definition is a mere declaration. The actual code of the function defining the actions performed by the function is found elsewhere (the concept of functions inside structs is further discussed in section 3.2).

- The size of the struct Point is equal to the size of its two ints. A function declared inside the structure does not affect its size. The compiler implements this behavior by allowing the function draw to be available only in the context of a Point.
The `Point` structure could be used as follows:

```c
Point a; // two points on
Point b; // the screen
a.x = 0; // define first dot
a.y = 10; // and draw it
a.draw();

b = a; // copy a to b
b.y = 20; // redefine y-coord
b.draw(); // and draw it
```

As shown in the above example a function that is part of the structure may be selected using the dot (.) (the arrow (->) operator is used when pointers to objects are available). This is therefore identical to the way data fields of structures are selected.

The idea behind this syntactic construction is that several types may contain functions having identical names. E.g., a structure representing a circle might contain three `int` values: two values for the coordinates of the center of the circle and one value for the radius. Analogously to the `Point` structure, a `Circle` may now have a function `draw` to draw the circle.
Chapter 3

A First Impression Of C++

In this chapter C++ is further explored. The possibility to declare functions in structs is illustrated in various examples; the concept of a class is introduced; casting is covered in detail; many new types are introduced and several important notational extensions to C are discussed.

3.1 Extensions to C

Before we continue with the ‘real’ object-approach to programming, we first introduce some extensions to the C programming language: not mere differences between C and C++, but syntactic constructs and keywords not found in C.

3.1.1 Namespaces

C++ introduces the notion of a namespace: all symbols are defined in a larger context, called a namespace. Namespaces are used to avoid name conflicts that could arise when a programmer would like to define a function like sin operating on degrees, but does not want to lose the capability of using the standard sin function, operating on radians.

Namespaces are covered extensively in chapter 4. For now it should be noted that most compilers require the explicit declaration of a standard namespace: std. So, unless otherwise indicated, it is stressed that all examples in the Annotations now implicitly use the

using namespace std;

declaration. So, if you actually intend to compile examples given in the C++ Annotations, make sure that the sources start with the above using declaration.

3.1.2 The scope resolution operator ::

C++ introduces several new operators, among which the scope resolution operator (::). This operator can be used in situations where a global variable exists having the same name as a local variable:

#include <stdio.h>
double counter = 50;  // global variable

int main()
{
    for (int counter = 1;  // this refers to the
        counter != 10;  // local variable
        ++counter)
    {
        printf("%d\n",
                ::counter  // global variable
                /  // divided by
                counter);  // local variable
    }
}

In the above program the scope operator is used to address a global variable instead of the local variable having the same name. In C++ the scope operator is used extensively, but it is seldom used to reach a global variable shadowed by an identically named local variable. Its main purpose is encountered in chapter 7.

### 3.1.3 Using the keyword ‘const’

Even though the keyword `const` is part of the C grammar, its use is more important and much more common in C++ than it is in C.

The `const` keyword is a modifier stating that the value of a variable or of an argument may not be modified. In the following example the intent is to change the value of a variable `ival`, which fails:

```c
int main()
{
    int const ival = 3;  // a constant int
                        // initialized to 3
    ival = 4;  // assignment produces
                // an error message
}
```

This example shows how `ival` may be initialized to a given value in its definition; attempts to change the value later (in an assignment) are not permitted.

Variables that are declared `const` can, in contrast to C, be used to specify the size of an array, as in the following example:

```c
int const size = 20;
char buf[size];  // 20 chars big
```

Another use of the keyword `const` is seen in the declaration of pointers, e.g., in pointer-arguments. In the declaration

```c
char const *buf;
```
buf is a pointer variable pointing to char. Whatever is pointed to by buf may not be changed through buf; the chars are declared as const. The pointer buf itself however may be changed. A statement like *buf = 'a'; is therefore not allowed, while ++buf is.

In the declaration

```c
char *const buf;
```

buf itself is a const pointer which may not be changed. Whatever chars are pointed to by buf may be changed at will.

Finally, the declaration

```c
char const *const buf;
```

is also possible; here, neither the pointer nor what it points to may be changed.

The rule of thumb for the placement of the keyword const is the following: whatever occurs to the left to the keyword may not be changed.

Although simple, this rule of thumb is often used. For example, Bjarne Stroustrup states (in http://www.research.att.com/~bs/bs_faq2.html#constplacement):

**Should I put "const" before or after the type?**

* I put it before, but that's a matter of taste. "const T" and "T const" were always (both) allowed and equivalent. For example:

```c
const int a = 1;       // OK
int const b = 2;       // also OK
```

* My guess is that using the first version will confuse fewer programmers ("is more idiomatic").

But we’ve already seen an example where applying this simple ‘before’ placement rule for the keyword const produces unexpected (i.e., unwanted) results as we will shortly see (below). Furthermore, the ‘idiomatic’ before-placement also conflicts with the notion of const functions, which we will encounter in section 7.7. With const functions the keyword const is also placed behind rather than before the name of the function.

The definition or declaration (either or not containing const) should always be read from the variable or function identifier back to the type indentifer:

* “Buf is a const pointer to const characters”

This rule of thumb is especially useful in cases where confusion may occur. In examples of C++ code published in other places one often encounters the reverse: const preceding what should not be altered. That this may result in sloppy code is indicated by our second example above:

```c
char const *buf;
```

What must remain constant here? According to the sloppy interpretation, the pointer cannot be altered (as const precedes the pointer). In fact, the char values are the constant entities here, as becomes clear when we try to compile the following program:

```c
int main()
```
{  
    char const *buf = "hello";
    ++buf; // accepted by the compiler
    *buf = 'u'; // rejected by the compiler
}

Compilation fails on the statement *buf = 'u'; and not on the statement ++buf.

Marshall Cline's C++ FAQ\(^1\) gives the same rule (paragraph 18.5), in a similar context:

\[18.5\] What's the difference between "const Fred* p", "Fred* const p" and "const Fred* const p"?
You have to read pointer declarations right-to-left.

Marshall Cline's advice can be improved, though. Here's a recipe that will effortlessly dissect even the most complex declaration:

1. start reading at the variable's name
2. read as far as possible until you reach the end of the declaration or an (as yet unmatched) closing parenthesis.
3. return to the point where you started reading, and read backwards until you reach the beginning of the declaration or a matching opening parenthesis.
4. If you reached an opening parenthesis, continue at step 2 beyond the parenthesis where you previously stopped.

Let's apply this recipe to the following (by itself irrelevant) complex declaration. Little arrows indicate how far we should read at each step and the direction of the arrow indicates the reading direction:

```
char const *(*(const (*(*ip)())[]))[]
```

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start at the variable’s name: 'ip' is</td>
</tr>
<tr>
<td>2</td>
<td>Hitting a closing paren: revert</td>
</tr>
<tr>
<td>3</td>
<td>Find the matching open paren: 'a pointer to'</td>
</tr>
<tr>
<td>4</td>
<td>The next unmatched closing par: 'a function (not expecting arguments)'</td>
</tr>
<tr>
<td>5</td>
<td>Find the matching open paren: 'returning a pointer to'</td>
</tr>
<tr>
<td>6</td>
<td>The next closing par:</td>
</tr>
</tbody>
</table>

\(^1\)http://www.parashift.com/c++-faq-lite/const-correctness.html
3.1. EXTENSIONS TO C

--> 'an array of'

(* const (.(*ip)())[]) Find the matching open paren:
<------- 'const pointers to'

(* const (.(*ip)())[])[][] Read until the end:
-> 'an array of'

char const *(* const (*(*ip)())[])[] Read backwards what’s left:
<---------- 'pointers to const chars'

Collecting all the parts, we get for char const *(* const (*(*ip)())[])[]: ip is a pointer to a function (not expecting arguments), returning a pointer to an array of const pointers to an array of pointers to const chars. This is what ip represents; the recipe can be used to parse any declaration you ever encounter.

3.1.4 ‘cout’, ‘cin’, and ‘cerr’

Analogous to C, C++ defines standard input- and output streams which are available when a program is executed. The streams are:

- cout, analogous to stdout,
- cin, analogous to stdin,
- cerr, analogous to stderr.

Syntactically these streams are not used as functions: instead, data are written to streams or read from them using the operators <<, called the insertion operator and >>, called the extraction operator. This is illustrated in the next example:

```cpp
#include <iostream>

using namespace std;

int main()
{
    int ival;
    char sval[30];

    cout << "Enter a number:\n";
    cin >> ival;
    cout << "And now a string:\n";
    cin >> sval;

    cout << "The number is: " << ival << "\n"
         "And the string is: " << sval << '\n';
}
```

This program reads a number and a string from the cin stream (usually the keyboard) and prints these data to cout. With respect to streams, please note:
• The standard streams are declared in the header file iostream. In the examples in the C++ Annotations this header file is often not mentioned explicitly. Nonetheless, it must be included (either directly or indirectly) when these streams are used. Comparable to the use of the using namespace std; clause, the reader is expected to #include <iostream> with all the examples in which the standard streams are used.

• The streams cout, cin and cerr are variables of so-called class-types. Such variables are commonly called objects. Classes are discussed in detail in chapter 7 and are used extensively in C++.

• The stream cin extracts data from a stream and copies the extracted information to variables (e.g., ival in the above example) using the extraction operator (two consecutive > characters: >>). Later in the Annotations we will describe how operators in C++ can perform quite different actions than what they are defined to do by the language, as is the case here. Function overloading has already been mentioned. In C++ operators can also have multiple definitions, which is called operator overloading.

• The operators which manipulate cin, cout and cerr (i.e., >> and <<) also manipulate variables of different types. In the above example cout << ival results in the printing of an integer value, whereas cout << "Enter a number" results in the printing of a string. The actions of the operators therefore depend on the types of supplied variables.

• The extraction operator (>>) performs a so called type safe assignment to a variable by 'extracting' its value from a text stream. Normally, the extraction operator skips all white space characters preceding the values to be extracted.

• Special symbolic constants are used for special situations. Normally a line is terminated by inserting "\n" or '\n'. But when inserting the endl symbol the line is terminated followed by the flushing of the stream's internal buffer. Thus, endl can usually be avoided in favor of '\n' resulting in somewhat more efficient code.

The stream objects cin, cout and cerr are not part of the C++ grammar proper. The streams are part of the definitions in the header file iostream. This is comparable to functions like printf that are not part of the C grammar, but were originally written by people who considered such functions important and collected them in a run-time library.

A program may still use the old-style functions like printf and scanf rather than the new-style streams. The two styles can even be mixed. But streams offer several clear advantages and in many C++ programs have completely replaced the old-style C functions. Some advantages of using streams are:

• Using insertion and extraction operators is type-safe. The format strings which are used with printf and scanf can define wrong format specifiers for their arguments, for which the compiler sometimes can’t warn. In contrast, argument checking with cin, cout and cerr is performed by the compiler. Consequently it isn't possible to err by providing an int argument in places where, according to the format string, a string argument should appear. With streams there are no format strings.

• The functions printf and scanf (and other functions using format strings) in fact implement a mini-language which is interpreted at run-time. In contrast, with streams the C++ compiler knows exactly which in- or output action to perform given the arguments used. No mini-language here.

• In addition the possibilities of the insertion and extraction operators may be extended allowing objects of classes that didn't exist when the streams were originally designed to be inserted into or extracted from streams. Mini languages as used with printf cannot be extended.
3.2. Functions as part of structs

- The usage of the left-shift and right-shift operators in the context of streams illustrates yet another capability of C++: operator overloading allowing us to redefine the actions an operator performs in certain contexts. Ascending from C operator overloading requires some getting used, but after a short little while these overloaded operators feel rather comfortable.

- Streams are independent of the media they operate upon. This (at this point somewhat abstract) notion means that the same code can be used without any modification at all to interface your code to any kind of device. The code using streams can be used when the device is a file on disk; an Internet connection; a digital camera; a DVD device; a satellite link; and much more: you name it. Streams allow your code to be decoupled (independent) of the devices your code is supposed to operate on, which eases maintenance and allows reuse of the same code in new situations.

The iostream library has a lot more to offer than just cin, cout and cerr. In chapter 6 iostreams are covered in greater detail. Even though printf and friends can still be used in C++ programs, streams have practically replaced the old-style C I/O functions like printf. If you think you still need to use printf and related functions, think again: in that case you’ve probably not yet completely grasped the possibilities of stream objects.

3.2 Functions as part of structs

Earlier it was mentioned that functions can be part of structs (see section 2.5.13). Such functions are called member functions. This section briefly discusses how to define such functions.

The code fragment below shows a struct having data fields for a person’s name and address. A function print is included in the struct’s definition:

```
struct Person
{
    char name[80];
    char address[80];

    void print();
};
```

When defining the member function print the structure’s name (Person) and the scope resolution operator (::) are used:

```
void Person::print()
{
    cout << "Name: " << name << "\n"
    "Address: " << address << "\n";
}
```

The implementation of Person::print shows how the fields of the struct can be accessed without using the structure’s type name. Here the function Person::print prints a variable name. Since Person::print is itself a part of struct person, the variable name implicitly refers to the same type.

This struct Person could be used as follows:

```
Person person;
```
CHAPTER 3. A FIRST IMPRESSION OF C++

```c
strcpy(person.name, "Karel");
strcpy(person.address, "Marskramerstraat 33");
person.print();
```

The advantage of member functions is that the called function automatically accesses the data fields of the structure for which it was invoked. In the statement `person.print()` the object `person` is the 'substrate': the variables `name` and `address` that are used in the code of `print` refer to the data stored in the `person` object.

### 3.2.1 Data hiding: public, private and class

As mentioned before (see section 2.3), C++ contains specialized syntactic possibilities to implement data hiding. Data hiding is the capability of sections of a program to hide its data from other sections. This results in very clean data definitions. It also allows these sections to enforce the integrity of their data.

C++ has three keywords that are related to data hiding: **private**, **protected** and **public**. These keywords can be used in the definition of structs. The keyword **public** allows all subsequent fields of a structure to be accessed by all code; the keyword **private** only allows code that is part of the struct itself to access subsequent fields. The keyword **protected** is discussed in chapter 13, and is somewhat outside of the scope of the current discussion.

In a **struct** all fields are **public**, unless explicitly stated otherwise. Using this knowledge we can expand the **struct Person**:

```c
struct Person
{
    private:
        char d_name[80];
        char d_address[80];
    public:
        void setName(char const *n);
        void setAddress(char const *a);
        void print();
        char const *name();
        char const *address();
};
```

As the data fields `d_name` and `d_address` are in a **private** section they are only accessible to the member functions which are defined in the **struct**: these are the functions `setName`, `setAddress` etc.. As an illustration consider the following code:

```c
Person fbb;

fbb.setName("Frank");       // OK, setName is public
strcpy(fbb.d_name, "Knarf"); // error, \texttt{x.d\_name} is private
```

Data integrity is implemented as follows: the actual data of a struct `Person` are mentioned in the structure definition. The data are accessed by the outside world using special functions that are also part of the definition. These member functions control all traffic between the data fields and other parts of the program and are therefore also called 'interface' functions. The thus implemented
data hiding is illustrated in Figure 3.1. The members `setName` and `setAddress` are declared with `char const *` parameters. This indicates that the functions will not alter the strings which are supplied as their arguments. Analogously, the members `name` and `address` return `char const *`: the compiler prevents callers of those members from modifying the information made accessible through the return values of those members.

Two examples of member functions of the `struct Person` are shown below:

```cpp
void Person::setName(char const *n)  
{  
    strncpy(d_name, n, 79);  
    d_name[79] = 0;  
}  

char const *Person::name()  
{  
    return d_name;  
}
```

The power of member functions and of the concept of data hiding results from the abilities of member functions to perform special tasks, e.g., checking the validity of the data. In the above example `setName` copies only up to 79 characters from its argument to the data member `name`, thereby avoiding a buffer overflow.

Another illustration of the concept of data hiding is the following. As an alternative to member functions that keep their data in memory a library could be developed featuring member functions storing data on file. To convert a program storing `Person` structures in memory to one that stores the data on disk no special modifications are required. After recompilation and linking the program to a new library it is converted from storage in memory to storage on disk. This example illustrates a broader concept than data hiding; it illustrates encapsulation. Data hiding is a kind of encapsulation. Encapsulation in general results in reduced coupling of different sections of a program. This in turn greatly enhances reusability and maintainability of the resulting software. By having the structure encapsulate the actual storage medium the program using the structure becomes independent of the actual storage medium that is used.

Though data hiding can be implemented using `structs`, more often (almost always) `classes` are
used instead. A class is a kind of struct, except that a class uses private access by default, whereas structs use public access by default. The definition of a class `Person` is therefore identical to the one shown above, except for the fact that the keyword `class` has replaced `struct` while the initial `private:` clause can be omitted. Our typographic suggestion for class names (and other type names defined by the programmer) is to start with a capital character to be followed by the remainder of the type name using lower case letters (e.g., `Person`).

### 3.2.2 Structs in C vs. structs in C++

In this section we'll discuss an important difference between C and C++ structs and (member) functions. In C it is common to define several functions to process a `struct`, which then require a pointer to the `struct` as one of their arguments. An imaginary C header file showing this concept is:

```c
/* definition of a struct PERSON This is C */
typedef struct
{
    char name[80];
    char address[80];
} PERSON;

/* some functions to manipulate PERSON structs */

/* initialize fields with a name and address */
void initialize(PERSON *p, char const *nm, char const *adr);

/* print information */
void print(PERSON const *p);

/* etc. */
```

In C++, the declarations of the involved functions are put inside the definition of the `struct` or `class`. The argument denoting which `struct` is involved is no longer needed.

```c++
class Person
{
    char d_name[80];
    char d_address[80];

    public:
        void initialize(char const *nm, char const *adr);
        void print();
        // etc..
};
```

In C++ the `struct` parameter is not used. A C function call such as:

```c
PERSON x;
initialize(&x, "some name", "some address");
```

becomes in C++:
Person x;

x.initialize("some name", "some address");

### 3.3 More extensions to C

#### 3.3.1 References

In addition to the common ways to define variables (plain variables or pointers) C++ introduces **references** defining synonyms for variables. A reference to a variable is like an alias; the variable and the reference can both be used in statements involving the variable:

```
int int_value;
int &ref = int_value;
```

In the above example a variable `int_value` is defined. Subsequently a reference `ref` is defined, which (due to its initialization) refers to the same memory location as `int_value`. In the definition of `ref`, the reference operator `&` indicates that `ref` is not itself an `int` but a reference to one. The two statements

```
++int_value;
++ref;
```

have the same effect: they increment `int_value`'s value. Whether that location is called `int_value` or `ref` does not matter.

References serve an important function in C++ as a means to pass modifiable arguments to functions. E.g., in standard C, a function that increases the value of its argument by five and returning nothing needs a pointer parameter:

```c
void increase(int *valp) // expects a pointer
{ // to an int
    *valp += 5;
}

int main()
{
    int x;
    increase(&x); // pass x’s address
}
```

This construction can also be used in C++ but the same effect is also achieved using a reference:

```c
void increase(int &valr) // expects a reference
{ // to an int
    valr += 5;
}

int main()


{  
    int x;  

    increase(x);  // passed as reference  
}

It is arguable whether code such as the above should be preferred over C's method, though. The statement \texttt{increase(x)} suggests that not \texttt{x} itself but a \textit{copy} is passed. Yet the value of \texttt{x} changes because of the way \texttt{increase()} is defined. However, references can also be used to pass objects that are only inspected (without the need for a copy or a const *) or to pass objects whose modification is an accepted side-effect of their use. In those cases using references are strongly preferred over existing alternatives like copy by value or passing pointers.

Behind the scenes references are implemented using pointers. So, as far as the compiler is concerned references in C++ are just const pointers. With references, however, the programmer does not need to know or to bother about levels of indirection. An important distinction between plain pointers and references is of course that with references no indirection takes place. For example:

\begin{verbatim}
extern int *ip;
extern int &ir;

ip = 0;  // reassigns ip, now a 0-pointer
ir = 0;  // ir unchanged, the int variable it refers to
         // is now 0.
\end{verbatim}

In order to prevent confusion, we suggest to adhere to the following:

- In those situations where a function does not alter its parameters of a built-in or pointer type, value parameters can be used:

\begin{verbatim}
void some_func(int val)
{
    cout << val << \’n\’;
}

int main()
{
    int x;
    
    some_func(x);  // a copy is passed
}
\end{verbatim}

- When a function explicitly must change the values of its arguments, a pointer parameter is preferred. These pointer parameters should preferably be the function’s initial parameters. This is called \textit{return by argument}.

\begin{verbatim}
void by_pointer(int *valp)
{
    *valp += 5;
}
\end{verbatim}

- When a function doesn’t change the value of its class- or struct-type arguments, or if the modification of the argument is a trivial side-effect (e.g., the argument is a stream) references can be used. Const-references should be used if the function does not modify the argument:

\begin{verbatim}
void by_reference(string const &str)
\end{verbatim}
{  
cout << str; // no modification of str
}

int main ()
{
    int x = 7;
    by_pointer(&x); // a pointer is passed
    // x might be changed
    string str("hello");
    by_reference(str); // str is not altered
}

References play an important role in cases where the argument is not changed by the function but where it is undesirable to copy the argument to initialize the parameter. Such a situation occurs when a large object is passed as argument, or is returned by the function. In these cases the copying operation tends to become a significant factor, as the entire object must be copied. In these cases references are preferred.

If the argument isn't modified by the function, or if the caller shouldn't modify the returned information, the const keyword should be used. Consider the following example:

```c
struct Person // some large structure
{
    char    name[80];
    char    address[90];
    double  salary;
};

Person person[50]; // database of persons

    // printperson expects a
    // reference to a structure
    // but won't change it
void printperson (Person const &p)
{
    cout << "Name: " << p.name << 'n' <<
            "Address: " << p.address << 'n';
}

    // get a person by indexvalue
Person const &person(int index)
{
    return person[index]; // a reference is returned,
} // not a copy of person[index]

int main()
{
    Person boss;

    printperson (boss); // no pointer is passed,
    // so variable won't be
    // altered by the function
    printperson(person(5)); // references, not copies
```
Furthermore, note that there is yet another reason for using references when passing objects as function arguments. When passing a reference to an object, the activation of a so called copy constructor is avoided. Copy constructors are covered in chapter 9.

References could result in extremely ‘ugly’ code. A function may return a reference to a variable, as in the following example:

```cpp
int &func()
{
    static int value;
    return value;
}
```

This allows the use of the following constructions:

```cpp
func() = 20;
func() += func();
```

It is probably superfluous to note that such constructions should normally not be used. Nonetheless, there are situations where it is useful to return a reference. We have actually already seen an example of this phenomenon in our previous discussion of streams. In a statement like `cout << "Hello" << '\n';` the insertion operator returns a reference to `cout`. So, in this statement first the "Hello" is inserted into `cout`, producing a reference to `cout`. Through this reference the ‘ \n’ is then inserted in the `cout` object, again producing a reference to `cout`, which is then ignored.

Several differences between pointers and references are pointed out in the next list below:

- A reference cannot exist by itself, i.e., without something to refer to. A declaration of a reference like
  ```cpp
  int &ref;
  ```
  is not allowed; what would `ref` refer to?
- References can be declared as `external`. These references were initialized elsewhere.
- References may exist as parameters of functions: they are initialized when the function is called.
- References may be used in the return types of functions. In those cases the function determines what the return value refers to.
- References may be used as data members of classes. We return to this usage later.
- Pointers are variables by themselves. They point at something concrete or just “at nothing”.
- References are aliases for other variables and cannot be re-aliased to another variable. Once a reference is defined, it refers to its particular variable.
- Pointers (except for `const` pointers) can be reassigned to point to different variables.
- When an address-of operator `&` is used with a reference, the expression yields the address of the variable to which the reference applies. In contrast, ordinary pointers are variables themselves, so the address of a pointer variable has nothing to do with the address of the variable pointed to.
### 3.3.2 Rvalue References

In C++, temporary (rvalue) values are indistinguishable from `const &` types. The C++11 standard adds a new reference type called an *rvalue reference*, defined as `typename &&`. The name *rvalue* reference is derived from assignment statements, where the variable to the left of the assignment operator is called an *lvalue* and the expression to the right of the assignment operator is called an *rvalue*. Rvalues are often temporary (or anonymous) values, like values returned by functions.

In this parlance the C++ reference should be considered an *lvalue reference* (using the notation `typename &`). They can be contrasted to *rvalue references* (using the notation `typename &&`).

The key to understanding rvalue references is the concept of an *anonymous variable*. An anonymous variable has no name and this is the distinguishing feature for the compiler to associate it automatically with an rvalue reference if it has a choice. Before introducing some interesting and new constructions that weren't available before C++11 let's first have a look at some standard situations where *lvalue* references are used. The following function returns a temporary (anonymous) value:

```cpp
int intVal()
{
    return 5;
}
```

Although `intVal`'s return value can be assigned to an `int` variable it requires copying, which might become prohibitive when a function does not return an `int` but instead some large object. A reference or pointer cannot be used either to collect the anonymous return value as the return value won't survive beyond that. So the following is illegal (as noted by the compiler):

```cpp
int &ir = intVal(); // fails: refers to a temporary
int const &ic = intVal(); // OK: immutable temporary
int *ip = &intVal(); // fails: no lvalue available
```

Apparently it is not possible to modify the temporary returned by `intVal`. But now consider these functions:

```cpp
void receive(int &value) // note: lvalue reference
{
    cout << "int value parameter\n";
}
void receive(int &&value) // note: rvalue reference
{
    cout << "int R-value parameter\n";
}
```

and let's call this function from `main`:

```cpp
int main()
{
    receive(18);
    int value = 5;
    receive(value);
```
This program produces the following output:

```
int R-value parameter
int value parameter
int R-value parameter
```

The program’s output shows the compiler selecting `receive(int &&value)` in all cases where it receives an anonymous `int` as its argument. Note that this includes `receive(18)`: a value 18 has no name and thus `receive(int &&value)` is called. Internally, it actually uses a temporary variable to store the 18, as is shown by the following example which modifies `receive`:

```
void receive(int &&value)
{
    ++value;
    cout << "int R-value parameter, now: " << value << '\n';
    // displays 19 and 6, respectively.
}
```

Contrasting `receive(int &value)` with `receive(int &&value)` has nothing to do with `int &value` not being a `const` reference. If `receive(int const &value)` is used the same results are obtained. Bottom line: the compiler selects the overloaded function using the rvalue reference if the function is passed an anonymous value.

The compiler runs into problems if `void receive(int &value)` is replaced by `void receive(int value)`, though. When confronted with the choice between a value parameter and a reference parameter (either lvalue or rvalue) it cannot make a decision and reports an ambiguity. In practical contexts this is not a problem. Rvalue references were added to the language in order to be able to distinguish the two forms of references: named values (for which lvalue references are used) and anonymous values (for which rvalue references are used).

It is this distinction that allows the implementation of move semantics and perfect forwarding. At this point the concept of move semantics cannot yet fully be discussed (but see section 9.7 for a more thorough discussion) but it is very well possible to illustrate the underlying ideas.

Consider the situation where a function returns a `struct Data` containing a pointer to dynamically allocated characters. Moreover, the struct defines a member function `copy(Data const &other)` that takes another `Data` object and copies the other’s data into the current object. The (partial) definition of the `struct Data` might look like this:\(^2\):

```
struct Data
{
    char *text;
    size_t size;
    void copy(Data const &other)
    {
        text = strdup(other.text);
        size = strlen(text);
    }
};
```

\(^2\)To the observant reader: in this example the memory leak that results from using `Data::copy()` should be ignored.
Next, functions `dataFactory` and `main` are defined as follows:

```c
Data dataFactory(char const *txt)
{
    Data ret = {strdup(txt), strlen(txt)};
    return ret;
}

int main()
{
    Data d1 = {strdup("hello"), strlen("hello")};
    Data d2;
    d2.copy(d1); // 1 (see text)
    Data d3;
    d3.copy(dataFactory("hello")); // 2
}
```

At (1) `d2` appropriately receives a copy of `d1`'s text. But at (2) `d3` receives a copy of the text stored in the temporary returned by the `dataFactory` function. As the temporary ceases to exist after the call to `copy()` two related and unpleasant consequences are observed:

- The return value is a temporary object: its only reason for existence is to pass its data on to `d3`. Now `d3` copies the temporary's data which clearly is somewhat overdone.
- The temporary `Data` object is lost following the call to `copy()`. Unfortunately its dynamically allocated data is lost as well resulting in a memory leak.

In cases like these `rvalue reference` should be used. By overloading the `copy` member with a member `copy(Data &&other)` the compiler is able to distinguish situations (1) and (2). It now calls the initial `copy()` member in situation (1) and the newly defined overloaded `copy()` member in situation (2):

```c
struct Data
{
    char *text;
    size_t size;
    void copy(Data const &other)
    {
        text = strdup(other.text);
    }
    void copy(Data &&other)
    {
        text = other.text;
        other.text = 0;
    }
};
```

Note that the overloaded `copy()` function merely moves the `other.text` pointer to the current object's `text` pointer followed by reassigning 0 to `other.text`. Struct `Data` suddenly has become `move-aware` and implements `move semantics`, removing the drawbacks of the previously shown approach:
• Instead of making a deep copy (which is required in situation (1)), the pointer value is simply moved to its new owner;

• Since the other.text doesn't point to dynamically allocated memory anymore the memory leak is prevented.

Rvalue references for *this and initialization of class objects by rvalues are not yet supported by the g++ compiler.

### 3.3.3 Strongly typed enumerations

Enumeration values in C++ are in fact int values, thereby bypassing type safety. E.g., values of different enumeration types may be compared for (in)equality, albeit through a (static) type cast.

Another problem with the current enum type is that their values are not restricted to the enum type name itself, but to the scope where the enumeration is defined. As a consequence, two enumerations having the same scope cannot have identical values.

In the C++11 standard these problems are solved by defining enum classes. An enum class can be defined as in the following example:

```cpp
group 1
enum class SafeEnum
{  
  NOT_OK,    // 0, by implication  
  OK = 10,  
  MAYBE_OK  // 11, by implication  
};
```

Enum classes use int values by default, but the used value type can easily be changed using the : type notation, as in:

```cpp
group 2
enum class CharEnum: unsigned char
{  
  NOT_OK,  
  OK
};
```

To use a value defined in an enum class its enumeration name must be provided as well. E.g., OK is not defined, CharEnum::OK is.

Using the data type specification (noting that it defaults to int) it is possible to use enum class forward declarations. E.g.,

```cpp
group 3
enum Enum1;    // Illegal: no size available  
enum Enum2: unsigned int;  // Legal in C++11: explicitly declared type

enum class Enum3;  // Legal in C++11: default int type is used  
enum class Enum4: char;  // Legal in C++11: explicitly declared type
```

A sequence of symbols of a strongly typed enumeration can also be indicated in a switch using the ellipsis syntax, as shown in the next example:

```cpp
SafeEnum enumValue();
```
switch (enumValue())
{
    case SafeEnum::NOT_OK ...
    case SafeEnum::OK:
        cout << "Status is known\n";
        break;

    default:
        cout << "Status unknown\n";
        break;
}

3.3.4 Initializer lists

The C language defines the initializer list as a list of values enclosed by curly braces, possibly themselves containing initializer lists. In C these initializer lists are commonly used to initialize arrays and structs.

C++ extends this concept in the C++11 standard by introducing the type

```
initializer_list<Type>
```

where Type is replaced by the type name of the values used in the initializer list. Initializer lists in C++ are, like their counterparts in C, recursive, so they can also be used with multi-dimensional arrays, structs and classes.

Before using the initializer_list the <initializer_list> header file must be included.

Like in C, initializer lists consist of a list of values surrounded by curly braces. But unlike C, functions can define initializer list parameters. E.g.,

```
void values(std::initializer_list<int> iniValues)
{
}
```

A function like values could be called as follows:

```
values({2, 3, 5, 7, 11, 13});
```

The initializer list appears as an argument which is a list of values surrounded by curly braces. Due to the recursive nature of initializer lists a two-dimensional series of values can also be passes, as shown in the next example:

```
void values2(std::initializer_list<std::initializer_list<int>> iniValues)
{
}
values2({{1, 2}, {2, 3}, {3, 5}, {4, 7}, {5, 11}, {6, 13}});
```

Initializer lists are constant expressions and cannot be modified. However, their size and values may be retrieved using their size, begin, and end members as follows:

```
void values(initializer_list<int> iniValues)
{
}
cout << "Initializer list having " << iniValues.size() << "values\n";
for
{
    initializer_list<int>::const_iterator begin = iniValues.begin();
    begin != iniValues.end();
    ++begin
}
cout << "Value: " << *begin << '\n';

Initializer lists can also be used to initialize objects of classes (cf. section 7.5).

### 3.3.5 Type inference using ‘auto’

A special use of the keyword `auto` is defined by the C++11 standard allowing the compiler to determine the type of a variable automatically rather than requiring the software engineer to define a variable’s type explicitly.

In parallel, the use of `auto` as a storage class specifier is no longer supported in the C++11 standard. According to that standard a variable definition like `auto int var` results in a compilation error.

This can be very useful in situations where it is very hard to determine the variable’s type in advance. These situations occur, e.g., in the context of templates, topics covered in chapters 18 until 23.

At this point in the Annotations only simple examples can be given. Also, some hints will be provided about more general uses of the `auto` keyword.

When defining and initializing a variable `int variable = 5` the type of the initializing expression is well known: it’s an `int`, and unless the programmer’s intentions are different this could be used to define `variable`’s type (although it shouldn’t in normal circumstances as it reduces rather than improves the clarity of the code):

    auto variable = 5;

Here are some examples where using `auto` is useful. In chapter 5 the `iterator` concept is introduced (see also chapters 12 and 18). Iterators sometimes have long type definitions, like

    std::vector<std::string>::const_reverse_iterator

Functions may return types like this. Since the compiler knows the types returned by functions we may exploit this knowledge using `auto`. Assuming that a function `begin()` is declared as follows:

    std::vector<std::string>::const_reverse_iterator begin();

Rather than writing the verbose variable definition (at `// 1`) a much shorter definition (at `// 2`) may be used:

    std::vector<std::string>::const_reverse_iterator iter = begin();  // 1
    auto iter = begin();    // 2
It’s easy to define additional variables of this type. When initializing those variables using `iter` the `auto` keyword can be used again:

```
auto start = iter;
```

If `start` can’t be initialized immediately using an existing variable the type of a well known variable or function can be used in combination with the `decltype` keyword, as in:

```
decltype(iter) start;
decltype(begin()) spare;
```

The keyword `decltype` may also receive an expression as its argument. This feature is already available in the C++11 standard implementation in g++ 4.3. E.g., `decltype(3 + 5)` represents an `int`, `decltype(3 / double(3))` represents `double`.

The `auto` keyword can also be used to postpone the definition of a function’s return type. The declaration of a function `intArrPtr` returning a pointer to an array of 10 `ints` looks like this:

```
int (*intArrPtr())[10];
```

Such a declaration is fairly complex. E.g., among other complexities it requires ‘protection of the pointer’ using parentheses in combination with the function’s parameter list. In situations like these the specification of the return type can be postponed using the `auto` return type, followed by the specification of the function’s return type after any other specification the function might receive (e.g., as a const member (cf. section 7.7) or following its `noexcept` specification (cf. section 23.7)).

Using `auto` to declare the above function, the declaration becomes:

```
auto intArrPtr() -> int (*)[10];
```

A return type specification using `auto` is called a `late-specified return type`.

The `auto` keyword can also be used to defined types that are related to the actual `auto` associated type. Here are some examples:

```
vector<int> vi;
auto iter = vi.begin(); // standard: auto is vector<int>::iterator
auto &&rref = vi.begin(); // auto is rvalue ref. to the iterator type
auto *ptr = &iter; // auto is pointer to the iterator type
auto *ptr = &rref; // same
```

### 3.3.6 Defining types and 'using' declarations

In C++ `typedef` is commonly used to define shorthand notations for complex types. Assume we want to define a shorthand for ‘a pointer to a function expecting a double and an int, and returning an unsigned long long int’. Such a function could be:

```
unsigned long long int compute(double, int);
```

A pointer to such a function has the following form:

```
unsigned long long int (*pf)(double, int);
```
If this kind of pointer is frequently used, consider defining it using `typedef`: simply put `typedef` in front of it and the pointer’s name is turned into the name of a type. It could be capitalized to let it stand out more clearly as the name of a type:

```cpp
typedef unsigned long long int (*PF)(double, int);
```

After having defined this type, it can be used to declare or define such pointers:

```cpp
PF pf = compute;  // initialize the pointer to a function like
                // 'compute'
void fun(PF pf);  // fun expects a pointer to a function like
                // 'compute'
```

However, including the pointer in the typedef might not be a very good idea, as it masks the fact that `pf` is a pointer. After all, `PF pf` looks more like `int x` than `int *x`. To document that `pf` is in fact a pointer, slightly change the `typedef`:

```cpp
typedef unsigned long long int FUN(double, int);
FUN *pf = compute;  // now pf clearly is a pointer.
```

The scope of typedefs is restricted to compilation units. Therefore, typedefs are usually embedded in header files which are then included by multiple source files in which the typedefs should be used.

In addition to `typedef` the C++11 standard offers the `using` keyword to associate a type and an identifier. In practice `typedef` and `using` can be used interchangeably. The `using` keyword arguably result in more readable type definitions. Consider the following three (equivalent) definitions:

- The traditional, C style definition of a type, embedding the type name in the definition (turning a variable name into a type name):

  ```cpp
typedef unsigned long long int FUN(double, int);
```

- Apply `using` to improve the visibility (for humans) of the type name, by moving the type name to the front of the definition:

  ```cpp
using FUN = unsigned long long int (double, int);
```

- An alternative construction, using a late-specified return type (cf. section 3.3.5):

  ```cpp
using FUN = auto (double, int) -> unsigned long long int;
```

### 3.3.7 Range-based for-loops

The C++ for-statement is identical to C’s for-statement:

```cpp
for (init; cond; inc)
  statement
```
Often the initialization, condition, and increment parts are fairly obvious, as in situations where all elements of an array or vector must be processed. Many languages offer the foreach statement for that and C++ offers the `std::for_each` generic algorithm (cf. section 19.1.17).

The C++11 standard adds new syntax for the `for`-statement: the `range-based for-loop`. This new syntax can be used to process all element of a range in turn. Three types of ranges are distinguished:

- Plain arrays (e.g., `int array[10]`);
- Initializer lists;
- Standard containers (or comparable) (cf. chapter 12);
- Any other type offering `begin()` and `end()` functions returning so-called `iterators` (cf. section 18.2).

In these cases the C++11 standard offers the following additional for-statement syntax:

```cpp
// assume int array[30]
for (auto &element: array)
    statement
```

The part to the left of the colon is called the `for range declaration`. The declared variable (element) is a formal name; use any identifier you like. The variable is only available within the nested statement, and it refers to (or is a copy of) each of the elements of the range, from the first element up to the last.

There's no formal requirement to use `auto`, but using `auto` is extremely useful in many situations. Not only in situations where the range refers to elements of some complex type, but also in situations where you know what you can do with the elements in the range, but don't care about their exact type names. In the above example `int` could also have been used.

The reference symbol (`&`) is important in the following cases:

- if you want to modify the elements in the nested statements
- if the elements themselves are structs (or classes, cf. chapter 7)

When the reference symbol is omitted the variable will be a copy of each of the subsequent elements of the range. Fine, probably, if you merely need to look at the variables when they are of primitive types, but needlessly inefficient if you have an array of `BigStruct` elements:

```cpp
struct BigStruct
{
    double array[100];
    int last;
};
```

Inefficient, because you don't need to make copies of the array's elements. Instead, use references to elements:

```cpp
BigStruct data[100]; // assume properly initialized elsewhere
int countUsed()
```cpp
{  
  int sum = 0;  
  // const &: the elements aren’t modified  
  for (auto const &element: data)  
    sum += element.last;  
  return sum;  
}
```

If `data` is only available as a pointer to its first element in combination with the number of elements, range-based for loops can also be used, but require a little help. Section 24.5 describes a generic approach to using range based for loops in such cases.

### 3.3.8 Raw String Literals

Standard ASCII-C strings are delimited by double quotes, supporting escape sequences like `\n`, `\\` and `\"`. In some cases it is useful to avoid escaping strings (e.g., in the context of XML). To this end, the C++11 standard offers *raw string literals*.

Raw string literals start with an `R`, followed by a double quote, followed by a label (which is an arbitrary sequence of characters not equal to `(`), followed by `)`. The raw string ends at the closing parenthesis `)`, followed by the label which is in turn followed by a double quote. Example:

```cpp
R"(A Raw \ "String")"  
R"delimiter(Another \ Raw \ "(String))delimiter"  
```

In the first case, everything between `"(` and `)` is part of the string. Escape sequences aren’t supported so the text `\"` within the first raw string literal defines three characters: a backslash, a blank character and a double quote. The second example shows a raw string defined between the markers "delimiter( and )delimiter".

### 3.3.9 Binary constants

In addition to hexadecimal integral constants (starting with `0x`), octal integral constants (starting with `0`), and decimal integral constants (starting with one of the digits 1..9), *binary* integral constants can be defined using the prefixes `0b` or `0B`. E.g., to represent the (decimal) value 5 the notation `0b101` can also be used.

The binary constants come in handy in the context of, e.g., bit-flags, as it immediately shows which bit-fields are set, while other notations are less informative.

### 3.4 New language-defined data types

In C the following built-in data types are available: `void`, `char`, `short`, `int`, `long`, `float` and `double`. C++ extends these built-in types with several additional built-in types: the types `bool`, `wchar_t`, `long long` and `long double` (Cf. ANSI/ISO draft (1995), par. 27.6.2.4.1 for examples of these very long types). The type `long long` is merely a double-long long datatype. The type `long double` is merely a double-long double datatype. These built-in types as well as pointer variables are called *primitive types* in the C++ Annotations.
There is a subtle issue to be aware of when converting applications developed for 32-bit architectures to 64-bit architectures. When converting 32-bit programs to 64-bit programs, only `long` types and pointer types change in size from 32 bits to 64 bits; integers of type `int` remain at their size of 32 bits. This may cause data truncation when assigning pointer or `long` types to `int` types. Also, problems with sign extension can occur when assigning expressions using types shorter than the size of an `int` to an `unsigned long` or to a pointer. More information about this issue can be found here\(^3\).

Except for these built-in types the class-type `string` is available for handling character strings. The datatypes `bool` and `wchar_t` are covered in the following sections, the datatype `string` is covered in chapter 5. Note that recent versions of C may also have adopted some of these newer data types (notably `bool` and `wchar_t`). Traditionally, however, C doesn’t support them, hence they are mentioned here.

Now that these new types are introduced, let’s refresh your memory about `letters` that can be used in `literal constants` of various types. They are:

- `b` or `B`: in addition to its use to indicate a hexadecimal value, it can also be used to define a `binary constant`. E.g., `0b101` equals the decimal value 5 (supported by, e.g., the g++ compiler, but not part of the C++11 standard).

- `E` or `e`: the `exponentiation` character in floating point literal values. For example: `1.23E+3`. Here, `E` should be pronounced (and interpreted) as: `times 10 to the power`. Therefore, `1.23E+3` represents the value 1230.

- `F` can be used as `postfix` to a non-integral numeric constant to indicate a value of type `float`, rather than `double`, which is the default. For example: `12.F` (the dot transforms 12 into a floating point value); `1.23E+3F` (see the previous example. `1.23E+3` is a `double` value, whereas `1.23E+3F` is a `float` value).

- `L` can be used as `prefix` to indicate a character string whose elements are `wchar_t`-type characters. For example: `L"hello world"`.

- `L` can be used as `postfix` to an integral value to indicate a value of type `long`, rather than `int`, which is the default. Note that there is no letter indicating a `short` type. For that a `static_cast<short>()` must be used.

- `p`, to specify the power in hexadecimal floating point numbers. E.g. `0x10p4`. The exponent itself is read as a decimal constant and can therefore not start with 0x. The exponent part is interpreted as a power of 2. So `0x10p2` is (decimal) equal to 64: `16 * 2^2`.

- `U` can be used as `postfix` to an integral value to indicate an `unsigned` value, rather than an `int`. It may also be combined with the `postfix` `L` to produce an `unsigned long int` value.

And, of course: the `x` and `a` until `f` characters can be used to specify hexadecimal constants (optionally using capital letters).

### 3.4.1 The data type ‘`bool`’

The type `bool` represents boolean (logical) values, for which the (now reserved) constants `true` and `false` may be used. Except for these reserved values, integral values may also be assigned to variables of type `bool`, which are then implicitly converted to `true` and `false` according to

---

\(^3\)http://developers.sun.com/solaris/articles/ILP32toLP64Issues.html
the following conversion rules (assume `intValue` is an int-variable, and `boolValue` is a bool-variable):

```cpp
// from int to bool:
boolValue = intValue ? true : false;

// from bool to int:
intValue = boolValue ? 1 : 0;
```

Furthermore, when `bool` values are inserted into streams then `true` is represented by 1, and `false` is represented by 0. Consider the following example:

```cpp
cout << "A true value: " << true << "\n"
    "A false value: " << false << '\n';
```

The `bool` data type is found in other programming languages as well. *Pascal* has its type `Boolean`; *Java* has a `boolean` type. Different from these languages, C++'s type `bool` acts like a kind of `int` type. It is primarily a documentation-improving type, having just two values `true` and `false`. Actually, these values can be interpreted as `enum` values for 1 and 0. Doing so would ignore the philosophy behind the `bool` data type, but nevertheless: assigning `true` to an `int` variable neither produces warnings nor errors.

Using the `bool`-type is usually clearer than using `int`. Consider the following prototypes:

```cpp
bool exists(char const *fileName); // (1)
int exists(char const *fileName);   // (2)
```

With the first prototype, readers expect the function to return `true` if the given filename is the name of an existing file. However, with the second prototype some ambiguity arises: intuitively the return value 1 is appealing, as it allows constructions like

```cpp
if (exists("myfile"))
    cout << "myfile exists";
```

On the other hand, many system functions (like `access`, `stat`, and many other) return 0 to indicate a successful operation, reserving other values to indicate various types of errors.

As a rule of thumb I suggest the following: if a function should inform its caller about the success or failure of its task, let the function return a `bool` value. If the function should return success or various types of errors, let the function return `enum` values, documenting the situation by its various symbolic constants. Only when the function returns a conceptually meaningful integral value (like the sum of two `int` values), let the function return an `int` value.

### 3.4.2 The data type ‘wchar_t’

The `wchar_t` type is an extension of the `char` built-in type, to accommodate *wide* character values (but see also the next section). The `g++` compiler reports `sizeof(wchar_t)` as 4, which easily accommodates all 65,536 different *Unicode* character values.

Note that *Java*’s `char` data type is somewhat comparable to C++’s `wchar_t` type. *Java*’s `char` type is 2 bytes wide, though. On the other hand, *Java*’s `byte` data type is comparable to C++’s `char` type: one byte. Confusing?
3.4. NEW LANGUAGE-DEFINED DATA TYPES

3.4.3 Unicode encoding

In C++ string literals can be defined as NTB strings. Prepending an NTBS by L (e.g., L"hello") defines a wchar_t string literal.

The new C++11 standard adds to this support for 8, 16 and 32 bit Unicode encoded strings. Furthermore, two new data types are introduced: char16_t and char32_t storing, respectively, a UTF-16 and UTF-32 unicode value.

In addition, a char type variable is large enough to contain any UTF-8 unicode value as well (i.e., it remains an 8-bit value).

String literals for the various types of unicode encodings (and associated variables) can be defined as follows:

```cpp
char utf_8[] = u8"This is UTF-8 encoded."
char16_t utf16[] = u"This is UTF-16 encoded."
char32_t utf32[] = U"This is UTF-32 encoded."
```

Alternatively, unicode constants may be defined using the \u escape sequence, followed by a hexadecimal value. Depending on the type of the unicode variable (or constant) a UTF-8, UTF-16 or UTF-32 value is used. E.g.,

```cpp
char utf_8[] = u8\\u2018
char16_t utf16[] = u\\u2018
char32_t utf32[] = U\\u2018
```

Unicode strings can be delimited by double quotes but raw string literals can also be used.

3.4.4 The data type ‘long long int’

The C++11 standard adds the type long long int to the set of standard types. On 32 bit systems it has at least 64 usable bits. Some compilers already supported long long int as an extension, but C++11 officially adds it to C++.

3.4.5 The data type ‘size_t’

The size_t type is not really a built-in primitive data type, but a data type that is promoted by POSIX as a typename to be used for non-negative integral values answering questions like ‘how much’ and ‘how many’, in which case it should be used instead of unsigned int. It is not a specific C++ type, but also available in, e.g., C. Usually it is defined implicitly when a (any) system header file is included. The header file ‘officially’ defining size_t in the context of C++ is cstddef.

Using size_t has the advantage of being a conceptual type, rather than a standard type that is then modified by a modifier. Thus, it improves the self-documenting value of source code.

Sometimes functions explicitly require unsigned int to be used. E.g., on amd-architectures the X-windows function XQueryPointer explicitly requires a pointer to an unsigned int variable as one of its arguments. In such situations a pointer to a size_t variable can’t be used, but the address of an unsigned int must be provided. Such situations are exceptional, though.
Other useful bit-represented types also exist. E.g., `uint32_t` is guaranteed to hold 32-bits unsigned values. Analogously, `int32_t` holds 32-bits signed values. Corresponding types exist for 8, 16 and 64 bits values. These types are defined in the header file `cstdint`.

### 3.5 A new syntax for casts

Traditionally, C offers the following cast syntax:

```
(typename)expression
```

Here `typename` is the name of a valid `type`, and `expression` is an expression.

C style casts are now deprecated. C++ programs should merely use the new style C++ casts as they offer the compiler facilities to verify the sensibility of the cast. Facilities which are not offered by the classic C-style cast.

A cast should not be confused with the often used constructor notation:

```
typename(expression)
```

the constructor notation is not a cast, but a request to the compiler to construct an (anonymous) variable of type `typename` from `expression`.

If casts are really necessary one of several new-style casts should be used. These new-style casts are introduced in the upcoming sections.

#### 3.5.1 The ‘static_cast’-operator

The `static_cast<type>(expression)` is used to convert ‘conceptually comparable or related types’ to each other. Here as well as in other C++ style casts `type` is the type to which the type of `expression` should be cast.

Here are some examples of situations where the `static_cast` can (or should) be used:

- **When converting an int to a double.**

  This happens, for example when the quotient of two `int` values must be computed without losing the fraction part of the division. The `sqrt` function called in the following fragment returns 2:

  ```
  int x = 19;
  int y = 4;
  sqrt(x / y);
  ```

  whereas it returns 2.179 when a `static_cast` is used, as in:

  ```
  sqrt(static_cast<double>(x) / y);
  ```

  The important point to notice here is that a `static_cast` is allowed to change the representation of its `expression` into the representation that’s used by the destination type.
Also note that the division is put outside of the cast expression. If the division is performed within the cast's expression (as in \texttt{static_cast\textless double\texttt{>>(x / y)}) an integer division has already been performed before the cast has had a chance to convert the type of an operand to double.

- **When converting enum values to int values (in any direction).**

Here the two types use identical representations, but different semantics. Assigning an ordinary \texttt{enum} value to an \texttt{int} doesn't require a cast, but when the enum is a strongly typed \texttt{enum} a cast is required. Conversely, a \texttt{static_cast} is required when assigning an \texttt{int} value to a variable of some enum type. Here is an example:

```cpp
enum class Enum {
    VALUE
};

cout << static_cast<int>(VALUE); // show the numeric value
```

- **When converting related pointers to each other.**

The \texttt{static_cast} is used in the context of class inheritance (cf. chapter 13) to convert a pointer to a so-called ‘derived class’ to a pointer to its ‘base class’. It cannot be used for casting unrelated types to each other (e.g., a \texttt{static_cast} cannot be used to cast a pointer to a short to a pointer to an int).

A \texttt{void *} is a generic pointer. It is frequently used by functions in the C library (e.g., \texttt{memcpy(3)}). Since it is the generic pointer it is related to any other pointer, and a \texttt{static_cast} should be used to convert a \texttt{void *} to an intended destination pointer. This is a somewhat awkward left-over from C, which should probably only be used in that context. Here is an example:

The \texttt{qsort} functions from the C library expects a pointer to a (comparison) function having two \texttt{void const *} parameters. In fact, these parameters point to data elements of the array to be sorted, and so the comparison function must cast the \texttt{void const *} parameters to pointers to the elements of the array to be sorted. So, if the array is an \texttt{int array[]} and the compare function's parameters are \texttt{void const *p1} and \texttt{void const *p2} then the compare function obtains the address of the \texttt{int} pointed to by \texttt{p1} by using:

```cpp
static_cast<int const *>(p1);
```

- **When undoing or introducing the signed-modifier of an int-typed variable (remember that a \texttt{static_cast} is allowed to change the expression's representation!).**

Here is an example: the C function \texttt{tolower} requires an \texttt{int} representing the value of an unsigned char. But char by default is a signed type. To call \texttt{tolower} using an available char \texttt{ch} we should use:

```cpp
tolower(static_cast<unsigned char>(ch))
```

### 3.5.2 The ‘const_cast’-operator

The \texttt{const} keyword has been given a special place in casting. Normally anything \texttt{const} is \texttt{const} for a good reason. Nonetheless situations may be encountered where the \texttt{const} can be ignored. For these special situations the \texttt{const_cast} should be used. Its syntax is:
const_cast<type>(expression)

A const_cast<type>(expression) expression is used to undo the const attribute of a (pointer) type.

The need for a const_cast may occur in combination with functions from the standard C library which traditionally weren’t always as const-aware as they should. A function `strfun(char *s)` might be available, performing some operation on its char *s parameter without actually modifying the characters pointed to by s. Passing `char const hello[] = "hello";` to `strfun` produces the warning

```
passing 'const char *' as argument 1 of 'fun(char *)' discards const
```

A const_cast is the appropriate way to prevent the warning:

```
strfun(const_cast<char *>(hello));
```

### 3.5.3 The ‘reinterpret_cast’-operator

The third new-style cast is used to change the interpretation of information: the reinterpret_cast. It is somewhat reminiscent of the static_cast, but reinterpret_cast should only be used when it is known that the information as defined in fact is or can be interpreted as something completely different. Its syntax is:

```
reinterpret_cast<pointer type>(pointer expression)
```

Think of the reinterpret_cast as a cast offering a poor-man’s union: the same memory location may be interpreted in completely different ways.

The reinterpret_cast is used, for example, in combination with the write function that is available for streams. In C++ streams are the preferred interface to, e.g., disk-files. The standard streams like `std::cin` and `std::cout` also are stream objects.

Streams intended for writing (‘output streams’ like cout) offer write members having the prototype

```
write(char const *buffer, int length)
```

To write the value stored within a double variable to a stream in its un-interpreted binary form the stream’s write member is used. However, as a double * and a char * point to variables using different and unrelated representations, a static_cast cannot be used. In this case a reinterpret_cast is required. To write the raw bytes of a variable double value to cout we use:

```
cout.write(reinterpret_cast<char const *>(value), sizeof(double));
```

All casts are potentially dangerous, but the reinterpret_cast is the most dangerous of them all. Effectively we tell the compiler: back off, we know what we’re doing, so stop fuzzing. All bets are off, and we’d better do know what we’re doing in situations like these. As a case in point consider the following code:

```
int value = 0x12345678; // assume a 32-bits int
```
3.5. A NEW SYNTAX FOR CASTS

The above code produces different results on little and big endian computers. Little endian computers show the value 78, big endian computers the value 12. Also note that the different representations used by little and big endian computers renders the previous example (cout.write(...)) non-portable over computers of different architectures.

As a rule of thumb: if circumstances arise in which casts have to be used, clearly document the reasons for their use in your code, making double sure that the cast does not eventually cause a program to misbehave. Also: avoid reinterpret_casts unless you have to use them.

3.5.4 The ‘dynamic_cast’-operator

Finally there is a new style cast that is used in combination with polymorphism (see chapter 14). Its syntax is:

```
    dynamic_cast<type>(expression)
```

Different from the static_cast, whose actions are completely determined compile-time, the dynamic_cast’s actions are determined run-time to convert a pointer to an object of some class (e.g., Base) to a pointer to an object of another class (e.g., Derived) which is found further down its so-called class hierarchy (this is also called downcasting).

At this point in the Annotations a dynamic_cast cannot yet be discussed extensively, but we return to this topic in section 14.6.1.

3.5.5 Casting 'shared_ptr' objects

In section 18.4 we’ll encounter the shared_ptr, defined by the C++11 standard.

To complete the overview of new-style casts, the casts to be used in combination with shared_ptr are mentioned below.

This section can safely be skipped without loss of continuity; actual coverage of these specialized casts is postponed until section 18.4.5.

These specialized casts are:

- static_pointer_cast, returning a shared_ptr to the base-class section of a derived class object;
- const_pointer_cast, returing a shared_ptr to a non-const object from a shared_ptr to a constant object;
- dynamic_pointer_cast, returning a shared_ptr to a derived class object from a shared_ptr to a base class object.
### 3.6 Keywords and reserved names in C++

C++'s keywords are a superset of C's keywords. Here is a list of all keywords of the language:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Keyword</th>
<th>Keyword</th>
<th>Keyword</th>
<th>Keyword</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignas</td>
<td>char32_t</td>
<td>enum</td>
<td>namespace</td>
<td>return</td>
<td>typedef</td>
</tr>
<tr>
<td>alignof</td>
<td>class</td>
<td>explicit</td>
<td>new</td>
<td>short</td>
<td>typeid</td>
</tr>
<tr>
<td>and</td>
<td>compl</td>
<td>export</td>
<td>noexcept</td>
<td>signed</td>
<td>typename</td>
</tr>
<tr>
<td>and_eq</td>
<td>concept</td>
<td>extern</td>
<td>not</td>
<td>sizeof</td>
<td>union</td>
</tr>
<tr>
<td>asm</td>
<td>const</td>
<td>false</td>
<td>not_eq</td>
<td>static</td>
<td>unsigned</td>
</tr>
<tr>
<td>auto</td>
<td>const_cast</td>
<td>float</td>
<td>nullptr</td>
<td>static_assert</td>
<td>using</td>
</tr>
<tr>
<td>axiom</td>
<td>constexpr</td>
<td>for</td>
<td>operator</td>
<td>static_cast</td>
<td>virtual</td>
</tr>
<tr>
<td>bitand</td>
<td>continue</td>
<td>friend</td>
<td>or</td>
<td>struct</td>
<td>void</td>
</tr>
<tr>
<td>bitor</td>
<td>decltype</td>
<td>goto</td>
<td>or_eq</td>
<td>switch</td>
<td>volatile</td>
</tr>
<tr>
<td>bool</td>
<td>default</td>
<td>if</td>
<td>private</td>
<td>template</td>
<td>wchar_t</td>
</tr>
<tr>
<td>break</td>
<td>delete</td>
<td>import</td>
<td>protected</td>
<td>this</td>
<td>while</td>
</tr>
<tr>
<td>case</td>
<td>do</td>
<td>inline</td>
<td>public</td>
<td>thread_local</td>
<td>xor</td>
</tr>
<tr>
<td>catch</td>
<td>double</td>
<td>int</td>
<td>register</td>
<td>throw</td>
<td>xor_eq</td>
</tr>
<tr>
<td>char</td>
<td>dynamic_cast</td>
<td>long</td>
<td>reinterpret_cast</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>char16_t</td>
<td>else</td>
<td>mutable</td>
<td>requires</td>
<td>try</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

- The `export` keyword is removed from the language under the C++11 standard, but remains a keyword, reserved for future use.

- The `operator` keywords: `and`, `and_eq`, `bitand`, `bitor`, `compl`, `not`, `not_eq`, `or`, `or_eq`, `xor` and `xor_eq` are symbolic alternatives for, respectively, `&&`, `&=`, `&`, `|`, `~`, `!`, `!=`, `||`, `|=`, `^`, `^=`.

- C++11 also recognizes the special identifiers `final` and `override`. These identifiers are special in the sense that they acquire special meanings when declaring classes or polymorphic functions. Section 14.4 provides further details.

Keywords can only be used for their intended purpose and cannot be used as names for other entities (e.g., variables, functions, class-names, etc.). In addition to keywords identifiers starting with an underscore and living in the *global namespace* (i.e., not using any explicit namespace or using the mere `::` namespace specification) or living in the *std namespace* are reserved identifiers in the sense that their use is a prerogative of the implementor.
Chapter 4

Name Spaces

4.1 Namespaces

Imagine a math teacher who wants to develop an interactive math program. For this program functions like \( \cos, \sin, \tan \) etc. are to be used accepting arguments in degrees rather than arguments in radians. Unfortunately, the function name \( \cos \) is already in use, and that function accepts radians as its arguments, rather than degrees.

Problems like these are usually solved by defining another name, e.g., the function name \( \text{cosDegrees} \) is defined. C++ offers an alternative solution through namespaces. Namespaces can be considered as areas or regions in the code in which identifiers may be defined. Identifiers defined in a namespace normally won't conflict with names already defined elsewhere (i.e., outside of their namespaces). So, a function \( \cos \) (expecting angles in degrees) could be defined in a namespace \( \text{Degrees} \). When calling \( \cos \) from within \( \text{Degrees} \) you would call the \( \cos \) function expecting degrees, rather than the standard \( \cos \) function expecting radians.

Now that the ANSI/ISO standard has been implemented to a large degree in recent compilers, the use of namespaces is more strictly enforced than in previous versions of compilers. This affects the setup of \texttt{class} header files. At this point in the Annotations this cannot be discussed in detail, but in section 7.12.1 the construction of header files using entities from namespaces is discussed.

4.1.1 Defining namespaces

Namespaces are defined according to the following syntax:

```cpp
namespace identifier
{
    // declared or defined entities
    // (declarative region)
}
```

The identifier used when defining a namespace is a standard C++ identifier.

Within the declarative region, introduced in the above code example, functions, variables, structs, classes and even (nested) namespaces can be defined or declared. Namespaces cannot be defined within a function body. However, it is possible to define a namespace using multiple namespace
declarations. Namespaces are ‘open’ meaning that a namespace CppAnnotations could be defined in a file file1.cc and also in a file file2.cc. Entities defined in the CppAnnotations namespace of files file1.cc and file2.cc are then united in one CppAnnotations namespace region. For example:

```cpp
// in file1.cc
namespace CppAnnotations
{
    double cos(double argInDegrees)
    {
        ...
    }
}

// in file2.cc
namespace CppAnnotations
{
    double sin(double argInDegrees)
    {
        ...
    }
}
```

Both sin and cos are now defined in the same CppAnnotations namespace.

Namespace entities can be defined outside of their namespaces. This topic is discussed in section 4.1.4.1.

### 4.1.1.1 Declaring entities in namespaces

Instead of defining entities in a namespace, entities may also be declared in a namespace. This allows us to put all the declarations in a header file that can thereupon be included in sources using the entities defined in the namespace. Such a header file could contain, e.g.,

```cpp
namespace CppAnnotations
{
    double cos(double degrees);
    double sin(double degrees);
}
```

### 4.1.1.2 A closed namespace

Namespaces can be defined without a name. Such an anonymous namespace restricts the visibility of the defined entities to the source file defining the anonymous namespace.

Entities defined in the anonymous namespace are comparable to C’s static functions and variables. In C++ the static keyword can still be used, but its preferred use is in class definitions (see chapter 7). In situations where in C static variables or functions would have been used the anonymous namespace should be used in C++.

The anonymous namespace is a closed namespace: it is not possible to add entities to the same anonymous namespace using different source files.
4.1.2 Referring to entities

Given a namespace and its entities, the scope resolution operator can be used to refer to its entities. For example, the function \texttt{cos()} defined in the \texttt{CppAnnotations} namespace may be used as follows:

```cpp
// assume CppAnnotations namespace is declared in the
// following header file:
#include <cppannotations>

int main()
{
    cout << "The cosine of 60 degrees is: " <<
         CppAnnotations::cos(60) << "\n";
}
```

This is a rather cumbersome way to refer to the \texttt{cos()} function in the \texttt{CppAnnotations} namespace, especially so if the function is frequently used. In cases like these an \textit{abbreviated} form can be used after specifying a \textit{using declaration}. Following

```cpp
using CppAnnotations::cos;  // note: no function prototype,
// just the name of the entity
// is required.
```

calling \texttt{cos} results in a call of the \texttt{cos} function defined in the \texttt{CppAnnotations} namespace. This implies that the standard \texttt{cos} function, accepting radians, is not automatically called anymore. To call that latter \texttt{cos} function the plain scope resolution operator should be used:

```cpp
int main()
{
    using CppAnnotations::cos;
    ...
    cout << cos(60)       // calls CppAnnotations::cos()
        << ::cos(1.5)    // call the standard cos() function
        << "\n";
}
```

A \textit{using} declaration can have restricted scope. It can be used inside a block. The \textit{using} declaration prevents the definition of entities having the same name as the one used in the \textit{using} declaration. It is not possible to specify a \textit{using} declaration for a variable \texttt{value} in some namespace, and to define (or declare) an identically named object in a block also containing a \textit{using} declaration. Example:

```cpp
int main()
{
    using CppAnnotations::value;
    ...
    cout << value << "\n";  // uses CppAnnotations::value
    int value;             // error: value already declared.
}
4.1.2.1 The ‘using’ directive

A generalized alternative to the using declaration is the using directive:

```cpp
using namespace CppAnnotations;
```

Following this directive, all entities defined in the CppAnnotations namespace are used as if they were declared by using declarations.

While the using directive is a quick way to import all the names of a namespace (assuming the namespace has previously been declared or defined), it is at the same time a somewhat dirty way to do so, as it is less clear what entity is actually used in a particular block of code.

If, e.g., cos is defined in the CppAnnotations namespace, CppAnnotations::cos is going to be used when cos is called. However, if cos is not defined in the CppAnnotations namespace, the standard cos function will be used. The using directive does not document as clearly as the using declaration what entity will actually be used. Therefore use caution when applying the using directive.

4.1.2.2 ‘Koenig lookup’

If Koenig lookup were called the ‘Koenig principle’, it could have been the title of a new Ludlum novel. However, it is not. Instead it refers to a C++ technicality.

‘Koenig lookup’ refers to the fact that if a function is called without specifying its namespace, then the namespaces of its argument types are used to determine the function’s namespace. If the namespace in which the argument types are defined contains such a function, then that function is used. This procedure is called the ‘Koenig lookup’.

As an illustration consider the next example. The function FBB::fun(FBB::Value v) is defined in the FBB namespace. It can be called without explicitly mentioning its namespace:

```cpp
#include <iostream>

namespace FBB
{
    enum Value // defines FBB::Value
    {
        FIRST
    };

    void fun(Value x)
    {
        std::cout << "fun called for " << x << '\n';
    }
}

int main()
{
    fun(FBB::FIRST); // Koenig lookup: no namespace
                     // for fun() specified
}
/*
The compiler is rather smart when handling namespaces. If `Value` in the namespace `FBB` would have been defined as `typedef int Value` then `FBB::Value` would be recognized as `int`, thus causing the Koenig lookup to fail.

As another example, consider the next program. Here two namespaces are involved, each defining their own `fun` function. There is no ambiguity, since the argument defines the namespace and `FBB::fun` is called:

```cpp
#include <iostream>

namespace FBB
{
  enum Value // defines FBB::Value
  {
    FIRST
  }

  void fun(Value x)
  {
    std::cout << "FBB::fun() called for " << x << '\n';
  }
}

namespace ES
{
  void fun(FBB::Value x)
  {
    std::cout << "ES::fun() called for " << x << '\n';
  }
}

int main()
{
  fun(FBB::FIRST); // No ambiguity: argument determines // the namespace
}
/*
generated output:
fun called for 0
*/
```

Here is an example in which there *is* an ambiguity: `fun` has two arguments, one from each namespace. The ambiguity must be resolved by the programmer:

```cpp
#include <iostream>

namespace ES
{
  enum Value // defines ES::Value
  {
```
An interesting subtlety with namespaces is that definitions in one namespace may break the code defined in another namespace. It shows that namespaces may affect each other and that namespaces may backfire if we're not aware of their peculiarities. Consider the following example:

```cpp
namespace FBB
{
    enum Value // defines FBB::Value
    {
        FIRST
    };

    void fun(Value x, ES::Value y)
    {
        std::cout << "FBB::fun() called\n";
    }
}

namespace ES
{
    void fun(FBB::Value x, Value y)
    {
        std::cout << "ES::fun() called\n";
    }
}

int main()
{
    // fun(FBB::FIRST, ES::FIRST); ambiguity: resolved by
    // explicitly mentioning
    // the namespace
    ES::fun(FBB::FIRST, ES::FIRST);
}

/*
generated output:
ES::fun() called
*/

namespace FBB
{
    struct Value
    {
    };

    void fun(int x);
    void gun(Value x);
}

namespace ES
{
    void fun(int x)
    {
```
Whatever happens, the programmer’d better not use any of the \texttt{ES::fun} functions since it results in infinite recursion. However, that’s not the point. The point is that the programmer won’t even be given the opportunity to call \texttt{ES::fun} since the compilation fails.

Compilation fails for \texttt{gun} but not for \texttt{fun}. But why is that so? Why is \texttt{ES::fun} flawlessly compiling while \texttt{ES::gun} isn’t? In \texttt{ES::fun fun(x)} is called. As \texttt{x’s type is not defined in a namespace the Koenig lookup does not apply and \texttt{fun calls itself with infinite recursion.}}

With \texttt{ES::gun} the argument is defined in the \texttt{FBB namespace}. Consequently, the \texttt{FBB::gun function is a possible candidate to be called. But \texttt{ES::gun itself also is possible as \texttt{ES::gun’s prototype perfectly matches the call \texttt{gun(x).}}}

Now consider the situation where \texttt{FBB::gun} has not yet been declared. Then there is of course no ambiguity. The programmer responsible for the \texttt{ES namespace} is resting happily. Some time after that, the programmer who’s maintaining the \texttt{FBB namespace} decides it may be nice to add a function \texttt{gun(Value x)} to the \texttt{FBB namespace}. Now suddenly the code in the namespace \texttt{ES breaks because of an addition in a completely other namespace (FBB). Namespaces clearly are not completely independent of each other and we should be aware of subtleties like the above. Later in the \texttt{C++ Annotations (chapter 11)} we’ll return to this issue.

### 4.1.3 The standard namespace

The \texttt{std} namespace is reserved by \texttt{C++}. The standard defines many entities that are part of the runtime available software (e.g., \texttt{cout, cin, cerr}); the templates defined in the \texttt{Standard Template Library} (cf. chapter 18); and the \texttt{Generic Algorithms} (cf. chapter 19) are defined in the \texttt{std namespace}.

Regarding the discussion in the previous section, \texttt{using declarations} may be used when referring to entities in the \texttt{std namespace}. For example, to use the \texttt{std::cout stream}, the code may declare this object as follows:

```cpp
#include <iostream>
using std::cout;
```

Often, however, the identifiers defined in the \texttt{std namespace} can all be accepted without much thought. Because of that, one frequently encounters a \texttt{using directive}, allowing the programmer to omit a namespace prefix when referring to any of the entities defined in the namespace specified with the \texttt{using directive}. Instead of specifying \texttt{using declarations} the following \texttt{using directive} is frequently encountered: construction like

```cpp
#include <iostream>
using namespace std;
```

Should a \texttt{using directive}, rather than \texttt{using declarations} be used? As a rule of thumb one might decide to stick to \texttt{using declarations}, up to the point where the list becomes impractically long, at which point a \texttt{using directive} could be considered.
Two restrictions apply to using directives and declarations:

- Programmers should not declare or define anything inside the namespace std. This is not compiler enforced but is imposed upon user code by the standard;
- Using declarations and directives should not be imposed upon code written by third parties. In practice this means that using directives and declarations should be banned from header files and should only be used in source files (cf. section 7.12.1).

### 4.1.4 Nesting namespaces and namespace aliasing

Namespaces can be nested. Here is an example:

```cpp
namespace CppAnnotations
{
    int value;
    namespace Virtual
    {
        void *pointer;
    }
}
```

The variable `value` is defined in the `CppAnnotations` namespace. Within the `CppAnnotations` namespace another namespace (`Virtual`) is nested. Within that latter namespace the variable `pointer` is defined. To refer to these variable the following options are available:

- The **fully qualified names** can be used. A fully qualified name of an entity is a list of all the namespaces that are encountered until reaching the definition of the entity. The namespaces and entity are glued together by the scope resolution operator:

  ```cpp
  int main()
  {
    CppAnnotations::value = 0;
    CppAnnotations::Virtual::pointer = 0;
  }
  ```

- A **using namespace CppAnnotations directive** can be provided. Now `value` can be used without any prefix, but `pointer` must be used with the `Virtual::` prefix:

  ```cpp
  using namespace CppAnnotations;
  int main()
  {
    value = 0;
    Virtual::pointer = 0;
  }
  ```

- A **using namespace directive for the full namespace chain** can be used. Now `value` needs its `CppAnnotations` prefix again, but `pointer` doesn't require a prefix anymore:

  ```cpp
  using namespace CppAnnotations::Virtual;
  ```
int main()
{
    CppAnnotations::value = 0;
    pointer = 0;
}

• When using two separate `using namespace` directives none of the namespace prefixes are required anymore:

```cpp
using namespace CppAnnotations;
using namespace Virtual;
```

```cpp
int main()
{
    value = 0;
    pointer = 0;
}
```

• The same can be accomplished (i.e., no namespace prefixes) for specific variables by providing specific `using declaration`:

```cpp
using CppAnnotations::value;
using CppAnnotations::Virtual::pointer;
```

```cpp
int main()
{
    value = 0;
    pointer = 0;
}
```

• A combination of `using namespace` directives and `using declaration` can also be used. E.g., a `using namespace` directive can be used for the `CppAnnotations::Virtual` namespace, and a `using declaration` can be used for the `CppAnnotations::value` variable:

```cpp
using namespace CppAnnotations::Virtual;
using CppAnnotations::value;
```

```cpp
int main()
{
    pointer = 0;
}
```

Following a `using namespace` directive all entities of that namespace can be used without any further prefix. If a single `using namespace` directive is used to refer to a nested namespace, then all entities of that nested namespace can be used without any further prefix. However, the entities defined in the more shallow namespace(s) still need the shallow namespace’s name(s). Only after providing specific `using namespace` directives or `using declaration` namespace qualifications can be omitted.

When fully qualified names are preferred but a long name like

```
CppAnnotations::Virtual::pointer
```

is considered too long, a `namespace alias` may be used:

```
namespace CV = CppAnnotations::Virtual;
```
This defines CV as an alias for the full name. The variable pointer may now be accessed using:

```cpp
CV::pointer = 0;
```

A namespace alias can also be used in a using namespace directive or using declaration:

```cpp
namespace CV = CppAnnotations::Virtual;
using namespace CV;
```

### 4.1.4.1 Defining entities outside of their namespaces

It is not strictly necessary to define members of namespaces inside a namespace region. But before an entity is defined outside of a namespace it must have been declared inside its namespace.

To define an entity outside of its namespace its name must be fully qualified by prefixing the member by its namespaces. The definition may be provided at the global level or at intermediate levels in the case of nested namespaces. This allows us to define an entity belonging to namespace A::B within the region of namespace A.

Assume the type `int INT8[8]` is defined in the `CppAnnotations::Virtual` namespace. Furthermore assume that it is our intent to define a function `squares`, inside the namespace `CppAnnotations::Virtual` returning a pointer to `CppAnnotations::Virtual::INT8`.

Having defined the prerequisites within the `CppAnnotations::Virtual` namespace, our function could be defined as follows (cf. chapter 9 for coverage of the memory allocation operator `new[]`):

```cpp
namespace CppAnnotations
{
    namespace Virtual
    {
        void *pointer;
        typedef int INT8[8];
        INT8 *squares()
        {
            INT8 *ip = new INT8[1];
            for (size_t idx = 0; idx != sizeof(INT8) / sizeof(int); ++idx)
                (*ip)[idx] = (idx + 1) * (idx + 1);
            return ip;
        }
    }
}
```

The function `squares` defines an array of one `INT8` vector, and returns its address after initializing the vector by the squares of the first eight natural numbers.

Now the function `squares` can be defined outside of the `CppAnnotations::Virtual` namespace:

```cpp
namespace CppAnnotations
{
```
4.1. NAMESPACES

```c
{
    namespace Virtual
    {
        void *pointer;
        typedef int INT8[8];
        INT8 *squares();
    }
}

CppAnnotations::Virtual::INT8 *CppAnnotations::Virtual::squares()
{
    INT8 *ip = new INT8[1];
    for (size_t idx = 0; idx != sizeof(INT8) / sizeof(int); ++idx)
        (*ip)[idx] = (idx + 1) * (idx + 1);
    return ip;
}
```

In the above code fragment note the following:

- `squares` is declared inside of the `CppAnnotations::Virtual` namespace.
- The definition outside of the namespace region requires us to use the fully qualified name of the function and its return type.
- Inside the body of the function `squares` we are within the `CppAnnotations::Virtual` namespace, so inside the function fully qualified names (e.g., for `INT8`) are not required any more.

Finally, note that the function could also have been defined in the `CppAnnotations` region. In that case the `Virtual` namespace would have been required when defining `squares()` and when specifying its return type, while the internals of the function would remain the same:

```c
namespace CppAnnotations
{
    namespace Virtual
    {
        void *pointer;
        typedef int INT8[8];
        INT8 *squares();
    }

    Virtual::INT8 *Virtual::squares()
    {
        INT8 *ip = new INT8[1];
        for (size_t idx = 0; idx != sizeof(INT8) / sizeof(int); ++idx)
            (*ip)[idx] = (idx + 1) * (idx + 1);
    }
```
return ip;
}
Chapter 5

The ‘string’ Data Type

C++ offers many solutions for common problems. Most of these facilities are part of the Standard Template Library or they are implemented as generic algorithms (see chapter 19).

Among the facilities C++ programmers have developed over and over again are those manipulating chunks of text, commonly called strings. The C programming language offers rudimentary string support. C’s NTBS is the foundation upon which an enormous amount of code has been built.

To process text C++ offers a std::string type. In C++ the traditional C library functions manipulating NTB strings are deprecated in favor of using string objects. Many problems in C programs are caused by buffer overruns, boundary errors and allocation problems that can be traced back to improperly using these traditional C string library functions. Many of these problems can be prevented using C++ string objects.

Actually, string objects are class type variables, and in that sense they are comparable to stream objects like cin and cout. In this section the use of string type objects is covered. The focus is on their definition and their use. When using string objects the member function syntax is commonly used:

```
stringVariable.operation(argumentList)
```

For example, if string1 and string2 are variables of type std::string, then

```
string1.compare(string2)
```

can be used to compare both strings.

In addition to the common member functions the string class also offers a wide variety of operators, like the assignment (=) and the comparison operator (==). Operators often result in code that is easy to understand and their use is generally preferred over the use of member functions offering comparable functionality. E.g., rather than writing

```
if (string1.compare(string2) == 0)
```

the following is generally preferred:

1. Following the C++11 standard, an NTBS (null-terminated byte string, also NTB string) is a character sequence whose highest-addressed element with defined content has the value zero (the terminating null character); no other character in the sequence has the value zero.
if (string1 == string2)

To define and use string-type objects, sources must include the header file `<string>`. To merely declare the string type the header can be included.

In addition to `std::string`, the header file `string` defines the following string types:

- `std::wstring`, a string type consisting of `wchar_t` characters;
- `std::u16string`, a string type consisting of `char16_t` characters;
- `std::u32string`, a string type consisting of `char32_t` characters.

## 5.1 Operations on strings

Some of the operations that can be performed on strings return indices within the strings. Whenever such an operation fails to find an appropriate index, the value `std::string::npos` is returned. This value is a symbolic value of type `std::string::size_type`, which is (for all practical purposes) an (unsigned) `int`.

All `std::string` members accepting `std::string` objects as arguments also accept `char const *` (NTBS) arguments. The same usually holds true for operators accepting `std::string` objects.

Some `std::string`-members use iterators. Iterators are formally introduced in section 18.2. Member functions using iterators are listed in the next section (5.2), but the iterator concept itself is not further covered by this chapter.

Strings support a large variety of members and operators. A short overview listing their capabilities is provided in this section, with subsequent sections offering a detailed discussion. The bottom line: C++ strings are extremely versatile and there is hardly a reason for falling back on the C library to process text. C++ strings handle all the required memory management and thus memory related problems, which is the #1 source of problems in C programs, can be prevented when C++ strings are used. Strings do come at a price, though. The class’s extensive capabilities have also turned it into a beast. It’s hard to learn and master all its features and in the end you’ll find that not all that you expected is actually there. For example, `std::string` doesn’t offer case-insensitive comparisons. But in the end it isn’t even as simple as that. It is there, but it is somewhat hidden and at this point in the C++ Annotations it’s too early to study into that hidden corner yet. Instead, realize that C’s standard library does offer useful functions that can be used as long as we’re aware of their limitations and are able to avoid their traps. So for now, to perform a traditional case-insensitive comparison of the contents of two `std::string` objects `str1` and `str2` the following will do:

```c++
strcasecmp(str1.c_str(), str2.c_str());
```

Strings support the following functionality:

- **Initialization:**
  
  when string objects are defined they are always properly initialized. In other words, they are always in a valid state. Strings may be initialized empty or already existing text can be used to initialize strings.

- **Assignment:**
  
  strings may be given new values. New values may be assigned using member functions (like `assign`) but a plain assignment operator (i.e., `=`) may also be used. Furthermore, assignment to a character buffer is also supported.
• **conversions:**

  the partial or complete contents of string objects may be interpreted as C strings but the string's contents may also be processed as a series of raw binary bytes, not necessarily terminating in a 0-valued character. Furthermore, in many situations plain characters and C strings may be used where `std::string` are accepted as well.

• **breakdown:**

  the individual characters stored in a string can be accessed using the familiar index operator ([ ]) allowing us to either access or modify information in the middle of a string.

• **comparisons:**

  strings may be compared to other strings (NTB strings) using the familiar logical comparison operators `==`, `!=`, `<`, `<=`, `>` and `>=`. There are also member functions available offering a more fine-grained comparison.

• **modification:**

  the contents of strings may be modified in many ways. Operators are available to add information to string objects, to insert information in the middle of string objects, or to replace or erase (parts of) a string's contents.

• **swapping:**

  the string's swapping capability allows us in principle to exchange the contents of two string objects without a byte-by-byte copying operation of the string's contents.

• **searching:**

  the locations of characters, sets of characters, or series of characters may be searched for from any position within the string object and either searching in a forward or backward direction.

• **housekeeping:**

  several housekeeping facilities are offered: the string's length, or its empty-state may be interrogated. But string objects may also be resized.

• **stream I/O:**

  strings may be extracted from or inserted into streams. In addition to plain string extraction a line of a text file may be read without running the risk of a buffer overrun. Since extraction and inserttion operations are stream based the I/O facilities are *device independent*.

## 5.2 A `std::string` reference

In this section the string members and string-related operations are referenced. The subsections cover, respectively the string's initializers, iterators, operators, and member functions. The following terminology is used throughout this section:

- **object** is always a `string-object`;
• argument is a string const & or a char const * unless indicated otherwise. The contents of an argument never is modified by the operation processing the argument;

• opos refers to an offset into an object string;

• apos refers to an offset into an argument;

• on represents a number of characters in an object (starting at opos);

• an represents a number of characters in an argument (starting at apos).

Both opos and apos must refer to existing offsets, or an exception (cf. chapter 10) is generated. In contrast, an and on may exceed the number of available characters, in which case only the available characters are considered.

Many members declare default values for on, an and apos. Some members declare default values for opos. Default offset values are 0, the default values of on and an is string::npos, which can be interpreted as ‘the required number of characters to reach the end of the string’.

With members starting their operations at the end of the string object’s contents proceeding backwards, the default value of opos is the index of the object’s last character, with on by default equal to opos + 1, representing the length of the substring ending at opos.

In the overview of member functions presented below it may be assumed that all these parameters accept default values unless indicated otherwise. Of course, the default argument values cannot be used if a function requires additional arguments beyond the ones otherwise accepting default values.

Some members have overloaded versions expecting an initial argument of type char const *. But even if that is not the case the first argument can always be of type char const * where a parameter of std::string is defined.

Several member functions accept iterators. Section 18.2 covers the technical aspects of iterators, but these may be ignored at this point without loss of continuity. Like apos and opos, iterators must refer to existing positions and/or to an existing range of characters within the string object’s contents.

Finally, all string-member functions computing indices return the predefined constant string::npos on failure.

### 5.2.1 Initializers

After defining string objects they are guaranteed to be in a valid state. At definition time string objects may be initialized in one of the following ways: The following string constructors are available:

• string object:
  
  initializes object to an empty string. When defining a string this way no argument list may be specified;

• string object(string::size_type count, char ch):
  
  initializes object with count characters ch;

• string object(string const &argument):
  
  initializes object with argument;
5.2. A \texttt{STD::STRING} Reference

- \texttt{string object(\texttt{std::string const \&argument, std::string::size_type apos, std::string::size_type an}):}
  - initializes \texttt{object} with \texttt{argument}'s contents starting at index position \texttt{apos}, using at most \texttt{an} of \texttt{argument}'s characters;
- \texttt{string object(InputIterator begin, InputIterator end):}
  - initializes \texttt{object} with the characters in the range of characters defined by the two \texttt{InputIterators}.

5.2.2 Iterators

See section 18.2 for details about iterators. As a quick introduction to iterators: an iterator acts like a pointer, and pointers can often be used in situations where iterators are requested. Iterators usually come in pairs, defining a range of entities. The begin-iterator points to the first entity, the end-iterator points just beyond the last entity of the range. Their difference is equal to the number of entities in the iterator-range.

Iterators play an important role in the context of generic algorithms (cf. chapter 19). The \texttt{std::string} defines the following \texttt{iterator types}:

- \texttt{string::iterator} and \texttt{string::const_iterator}:
  - these iterators are \texttt{forward iterators}. The \texttt{const_iterator} is returned by \texttt{string const} objects, the plain \texttt{iterator} is returned by non-const \texttt{string} objects. Characters referred to by iterators may be modified;
- \texttt{string::reverse_iterator} and \texttt{string::reverse_const_iterator}:
  - these iterators are also \texttt{forward iterators} but when incrementing the iterator the \texttt{previous} character in the \texttt{string} object is reached. Other than that they are comparable to, respectively, \texttt{string::iterator} and \texttt{string::const_iterator}.

5.2.3 Operators

String objects may be manipulated by member functions but also by operators. Using operators often results in more natural-looking code. In cases where operators are available having equivalent functionality as member function the operator is practically always preferred.

The following operators are available for \texttt{string} objects (in the examples 'object' and 'argument' refer to existing \texttt{std::string} objects).

- plain assignment:
  - a character, \texttt{C} or \texttt{C++} string may be assigned to a \texttt{string} object. The assignment operator returns its left-hand side operand. Example:
    ```
    object = argument;
    object = "C string";
    object = 'x';
    object = 120; // same as object = 'x'
    ```
CHAPTER 5. THE ‘STRING’ DATA TYPE

• addition:

the arithmetic additive assignment operator and the addition operator add text to a string object. The compound assignment operator returns its left-hand side operand, the addition operator returns its result in a temporary string object. When using the addition operator either the left-hand side operand or the right-hand side operand must be a std::string object. The other operand may be a char, a C string or a C++ string. Example:

```cpp
object += argument;
object += "hello";
object += 'x'; // integral expressions are OK

argument + otherArgument; // two std::string objects
argument + "hello"; // using + at least one
"hello" + argument; // std::string is required
argument + 'a'; // integral expressions are OK
'a' + argument;
```

• index operator:

The index operator may be used to retrieve object’s individual characters, or to assign new values to individual characters of a non-const string object. There is no range-checking (use the at() member function for that). This operator returns a char & or char const &. Example:

```cpp
object[3] = argument[5];
```

• logical operators:

the logical comparison operators may be applied to two string objects or to a string object and a C string to compare their contents. These operators return a bool value. The ==, !=, >, >=, <, and <= operators are available. The ordering operators perform a lexicographical comparison of their contents using the ASCII character collating sequence. Example:

```cpp
object == object; // true
object != (object + 'x'); // true
object <= (object + 'x'); // true
```

• stream related operators:

the insertion-operator (cf. section 3.1.4) may be used to insert a string object into an ostream, the extraction-operator may be used to extract a string object from an istream. The extraction operator by default first ignores all white space characters and then extracts all consecutively non-blank characters from an istream. Instead of a string a character array may be extracted as well, but the advantage of using a string object should be clear: the destination string object is automatically resized to the required number of characters. Example:

```cpp
cin >> object;
cout << object;
```

5.2.4 Member functions

The std::string class offers many member function as well as additional non-member functions that should be considered part of the string class. All these functions are listed below in alphabetic order.
5.2. A `STD::STRING` REFERENCE

The symbolic value `string::npos` is defined by the string class. It represents ‘index-not-found’ when returned by member functions returning string offset positions. Example: when calling `object.find('x')` (see below) on a string object not containing the character ‘x’, `npos` is returned, as the requested position does not exist.

The final 0-byte used in C strings to indicate the end of an NTBS is not considered part of a C++ string, and so the member function will return `npos`, rather than `length()` when looking for 0 in a string object containing the characters of a C string.

Here are the standard functions that operate on objects of the class string. When a parameter of `size_t` is mentioned it may be interpreted as a parameter of type `string::size_type`, but without defining a default argument value. The type `size_type` should be read as `string::size_type`. With `size_type` the default argument values mentioned in section 5.2 apply. All quoted functions are member functions of the class `std::string`, except where indicated otherwise.

- `char &at(size_t opos)`:
  
  a reference to the character at the indicated position is returned. When called with `string const` objects a `char const &` is returned. The member function performs range-checking, raising an exception (that by default aborts the program) if an invalid index is passed.

- `string &append(InputIterator begin, InputIterator end)`:
  
  the characters in the range defined by `begin` and `end` are appended to the current string object.

- `string &append(string const &argument, size_type apos, size_type an)`:
  
  argument (or a substring) is appended to the current string object.

- `string &append(char const *argument, size_type an)`:
  
  the first `an` characters of `argument` are appended to the string object.

- `string &append(size_type n, char ch)`:
  
  `n` characters `ch` are appended to the current string object.

- `string &assign(string const &argument, size_type apos, size_type an)`:
  
  argument (or a substring) is assigned to the string object. If `argument` is of type `char const *` and one additional argument is provided the second argument is interpreted as a value initializing `an`, using 0 to initialize `apos`.

- `string &assign(size_type n, char ch)`:
  
  `n` characters `ch` are assigned to the current string object.

- `char &back()`:
  
  returns a reference to the last `char` stored inside the string object. The result is undefined for empty strings.

- `string::iterator begin()`:
  
  an iterator referring to the first character of the current string object is returned. With `const string` objects a `const_iterator` is returned.
• `size_type capacity() const`:
  
  the number of characters that can currently be stored in the string object without needing to resize it is returned.

• `string::const_iterator cbegin()`:
  
  a `const_iterator` referring to the first character of the current string object is returned.

• `string::const_iterator cend()`:
  
  a `const_iterator` referring to the end of the current string object is returned.

• `int compare(string const &argument) const`:
  
  the text stored in the current string object and the text stored in `argument` is compared using a lexicographical comparison using the ASCII character collating sequence. zero is returned if the two strings have identical contents, a negative value is returned if the text in the current object should be ordered before the text in `argument`; a positive value is returned if the text in the current object should be ordered beyond the text in `argument`.

• `int compare(size_t opos, size_t on, string const &argument) const`:
  
  a substring of the text stored in the current string object is compared to the text stored in `argument`. At most `on` characters starting at offset `opos` are compared to the text in `argument`.

• `int compare(size_t opos, size_t on, string const &argument, size_type apos, size_type an)`:
  
  a substring of the text stored in the current string object is compared to a substring of the text stored in `argument`. At most `on` characters of the current string object, starting at offset `opos`, are compared to at most `an` characters of `argument`, starting at offset `apos`. In this case `argument` must be a string object.

• `int compare(size_t opos, size_t on, char const *argument, size_t an)`:
  
  a substring of the text stored in the current string object is compared to a substring of the text stored in `argument`. At most `on` characters of the current string object starting at offset `opos` are compared to at most `an` characters of `argument`. Argument must have at least `an` characters. The characters may have arbitrary values: 0-valued characters have no special meanings.

• `size_t copy(char *argument, size_t on, size_type opos) const`:
  
  the contents of the current string object are (partially) copied into `argument`. The actual number of characters copied is returned. The second argument, specifying the number of characters to copy, from the current string object is required. No 0-valued character is appended to the copied string but can be appended to the copied text using an idiom like the following:

  ```
  argument[object.copy(argument, string::npos)] = 0;
  ```

  Of course, the programmer should make sure that `argument`’s size is large enough to accomodate the additional 0-byte.

• `string::const_reverse_iterator cbegin()`:
  
  a `const_reverse_iterator` referring to the last character of the current string object is returned.
5.2. A STD::STRING REFERENCE

- string::const_reverse_iterator crend():
  a const_reverse_iterator referring to the begin of the current string object is returned.

- char const *c_str() const:
  the contents of the current string object as an NTBS.

- char const *data() const:
  the raw contents of the current string object are returned. Since this member does not return an NTBS (as c_str does), it can be used to retrieve any kind of information stored inside the current string object including, e.g., series of 0-bytes:

  ```cpp
  string s(2, 0);
  cout << static_cast<int>(s.data()[1]) << '\n';
  ```

- bool empty() const:
  true is returned if the current string object contains no data.

- string::iterator end():
  an iterator referring to the position just beyond the last character of the current string object is returned. With const string objects a const_iterator is returned.

- string &erase(size_type opos, size_type on):
  a (sub)string of the information stored in the current string object is erased.

- string::iterator erase(string::iterator begin, string::iterator end):
  the parameter end is optional. If omitted the value returned by the current object's end member is used. The characters defined by the begin and end iterators are erased. The iterator begin is returned, which is then referring to the position immediately following the last erased character.

- size_t find(string const &argument, size_type opos) const:
  the first index in the current string object where argument is found is returned.

- size_t find(char const *argument, size_type opos, size_type an) const:
  the first index in the current string object where argument is found is returned. When all three arguments are specified the first argument must be a char const *.

- size_t find(char ch, size_type opos) const:
  the first index in the current string object where ch is found is returned.

- size_t find_first_of(string const &argument, size_type opos) const:
  the first index in the current string object where any character in argument is found is returned.

- size_type find_first_of(char const *argument, size_type opos, size_type an) const:
  the first index in the current string object where any character in argument is found is returned. If opos is provided it refers to the first index in the current string object where the search for argument should start. If omitted, the string object is scanned completely. If an is provided it indicates the number of characters of the char const * argument that should be used in the search. It defines a substring starting at the beginning of argument. If omitted, all of argument's characters are used.
• `size_type find_first_of(char ch, size_type opos)`:  
  the first index in the current string object where character `ch` is found is returned.

• `size_t find_first_not_of(char ch, size_type opos) const`:  
  the first index in the current string object where another character than `ch` is found is returned.

• `size_t find_last_of(string const &argument, size_type opos) const`:  
  the last index in the current string object where any character in `argument` is found is returned.

• `size_t find_last_of(char const *argument, size_type opos, size_type an) const`:  
  the last index in the current string object where any character in `argument` is found is returned. If `opos` is provided it refers to the last index in the current string object where the search for `argument` should start. If omitted, the string object is scanned completely. If `an` is provided it indicates the number of characters of the `char const *` `argument` that should be used in the search. It defines a substring starting at the beginning of `argument`. If omitted, all of `argument`’s characters are used.

• `size_type find_last_of(char ch, size_type opos)`:  
  the last index in the current string object where character `ch` is found is returned.

• `size_t find_last_not_of(string const &argument, size_type opos) const`:  
  the last index in the current string object where any character not appearing in `argument` is found is returned.

• `char &front()`:  
  returns a reference to the first `char` stored inside the string object. The result is undefined for empty strings.

• `allocator_type get_allocator()`:  
  returns the allocator of the class `std::string`.

• `istream &std::getline(istream &istr, string &object, char delimiter = '\n')`:  
  Note: this is not a member function of the class `string`. A line of text is read from `istr`. All characters until `delimiter` (or the end of the stream, whichever comes first) are read from `istr` and are stored in `object`. If the delimiter is encountered it is removed from the stream, but is not stored in `line`. If the delimiter is not found, `istr.eof` returns `true` (see section 6.3.1). Since streams may be interpreted as `bool` values (cf. section 6.3.1) a commonly encountered idiom to read all lines from a stream successively into a string object `line` looks like this:

  ```
  while (getline(istr, line))
      process(line);
  ```

  The contents of the last line, whether or not it was terminated by a delimiter, is eventually also assigned to `object`. 
5.2. A STD::STRING REFERENCE

- string &insert(size_t opos, string const &argument, size_type apos, size_type an):
  a (sub)string of argument is inserted into the current string object at the current string object's index position opos. Arguments for apos and an must either both be provided or they must both be omitted.

- string &insert(size_t opos, char const *argument, size_type an):
  argument (of type char const *) is inserted at index opos into the current string object.

- string &insert(size_t opos, size_t count, char ch):
  Count characters ch are inserted at index opos into the current string object.

- string::iterator insert(string::iterator begin, char ch):
  the character ch is inserted at the current object's position referred to by begin. Begin is returned.

- string::iterator insert(string::iterator begin, size_t count, char ch):
  Count characters ch are inserted at the current object’s position referred to by begin. Begin is returned.

- string::iterator insert(string::iterator begin, InputIterator abegin, InputIterator aend):
  the characters in the range defined by the InputIterators abegin and aend are inserted at the current object's position referred to by begin. Begin is returned.

- size_t length() const:
  the number of characters stored in the current string object is returned.

- size_t max_size() const:
  the maximum number of characters that can be stored in the current string object is returned.

- void pop_back():
  The string's last character is removed from the string object.

- void push_back(char ch):
  The character ch is appended to the string object.

- void push_front(char ch):
  The character ch is prepended to the string object.

- string::reverse_iterator rbegin():
  a reverse iterator referring to the last character of the current string object is returned. With const string objects a reverse_const_iterator is returned.

- string::iterator rend():
  a reverse iterator referring to the position just before the first character of the current string object is returned. With const string objects a reverse_const_iterator is returned.
• string &replace(size_t opos, size_t on, string const &argument, size_type apos, size_type an):

  a (sub)string of characters in object are replaced by the (subset of) characters of argument. If on is specified as 0 argument is inserted into object at offset opos.

• string &replace(size_t opos, size_t on, char const *argument, size_type an):

  a series of characters in object are replaced by the first an characters of char const * argument.

• string &replace(size_t opos, size_t on, size_type count, char ch):

  on characters of the current string object, starting at index position opos, are replaced by count characters ch.

• string &replace(string::iterator begin, string::iterator end, string const &argument):

  the series of characters in the current string object defined by the iterators begin and end are replaced by argument. If argument is a char const *, an additional argument an may be used, specifying the number of characters of argument that are used in the replacement.

• string &replace(string::iterator begin, string::iterator end, size_type count, char ch):

  the series of characters in the current string object defined by the iterators begin and end are replaced by count characters having values ch.

• string &replace(string::iterator begin, string::iterator end, InputIterator abegin, InputIterator aend):

  the series of characters in the current string object defined by the iterators begin and end are replaced by the characters in the range defined by the InputIterators abegin and aend.

• void reserve(size_t request):

  the current string object’s capacity is changed to at least request. After calling this member, capacity’s return value will be at least request. A request for a smaller size than the value returned by capacity is ignored. A std::length_error exception is thrown if request exceeds the value returned by max_size (std::length_error is defined in the stdexcept header). Calling reserve() has the effect of redefining a string’s capacity, not of actually making available the memory to the program. This is illustrated by the exception thrown by the string’s at() member when trying to access an element exceeding the string’s size but not the string’s capacity.

• void resize(size_t size, char ch = 0):

  the current string object is resized to size characters. If the string object is resized to a size larger than its current size the additional characters will be initialized to ch. If it is reduced in size the characters having the highest indices are chopped off.

• size_t rfind(string const &argument, size_type opos) const:

  the last index in the current string object where argument is found is returned. Searching proceeds from the current object’s offset opos back to its beginning.
5.2. A STD::STRING REFERENCE

- **size_t rfind(char const *argument, size_type opos, size_type an) const**: 
  the last index in the current string object where argument is found is returned. 
  Searching proceeds from the current object’s offset opos back to its beginning. 
  The parameter an specifies the length of the substring of argument to look for, starting 
  at argument’s beginning.

- **size_t rfind(char ch, size_type opos)const**: 
  the last index in the current string object where ch is found is returned. 
  Searching proceeds from the current object’s offset opos back to its beginning.

- **void shrink_to_fit()**: 
  optionally reduces the amount of memory allocated by a vector to its current size. 
  The implementor is free to ignore or otherwise optimize this request. In order to 
  guarantee a ‘shrink to fit’ operation the 

    string(stringObject).swap(stringObject)

  idiom can be used.

- **size_t size() const**: 
  the number of characters stored in the current string object is returned. 
  This member is a synonym of length().

- **string substr(size_type opos, size_type on) const**: 
  a substring of the current string object of at most on characters starting at index 
  opos is returned.

- **void swap(string &argument)**: 
  the contents of the current string object are swapped with the contents of argument. 
  For this member argument must be a string object and cannot be a char const *.

5.2.5 Conversion functions

C++11 added several string conversion functions operating on or producing std::string objects. 
These functions are listed below in alphabetic order. They are not member functions, but class-less 
(free) functions declared in the std namespace. The <string> header file must be included before 
they can be used.

- **float stof(std::string const &str, size_t *pos = 0)**: 
  Initial white space characters in str are ignored. Then the following sequences of 
  characters are converted to a float value, which is returned:
  - A decimal floating point constant:
    - An optional + or - character
    - A series of decimal digits, possibly containing one decimal point character
    - An optional e or E character, followed by an optional - or + character, followed 
      by a series of decimal digits
  - A hexadecimal floating point constant:
    - An optional + or - character
    - 0x or 0X
CHAPTER 5. THE ‘STRING’ DATA TYPE

* A series of hexadecimal digits, possibly containing one decimal point character
* An optional p or P character, followed by an optional - or + character, followed by a series of decimal digits

- An infinity expression:
  * An optional + or - character
  * The words inf or infinity (case insensitive words)
- A ‘not a number’ expression:
  * An optional + or - character
  * The words nan or nan (alphanumeric character sequence) (nan is a case insensitive word), resulting in a NaN floating point value

If pos != 0 the index of the first character in str which was not converted is returned in *pos. A std::invalid_argument exception is thrown if the characters in str could not be converted to a float, a std::out_of_range exception is thrown if the converted value would have exceeded the range of float values.

• double stod(std::string const &str, size_t *pos = 0):

  A conversion as described with stof is performed, but now to a value of type double.

• double stold(std::string const &str, size_t *pos = 0):

  A conversion as described with stof is performed, but now to a value of type long double.

• int stoi(std::string const &str, size_t *pos = 0, int base = 10):

  Initial white space characters in str are ignored. Then all characters representing numeric constants of the number system whose base is specified are converted to an int value, which is returned. An optional + or - character may prefix the numeric characters. Values starting with 0 are automatically interpreted as octal values, values starting with 0x or 0X as hexadecimal characters. The value base must be between 2 and 36. If pos != 0 the index of the first character in str which was not converted is returned in *pos. A std::invalid_argument exception is thrown if the characters in str could not be converted to an int, a std::out_of_range exception is thrown if the converted value would have exceeded the range of int values.

  Here is an example of its use:

  ```
  int value = stoi(string(" -123")); // assigns value -123
  value = stoi(string(" 123", 0, 5)); // assigns value 38
  ```

• long stol(std::string const &str, size_t *pos = 0, int base = 10):

  A conversion as described with stoi is performed, but now to a value of type long.

• long long stoll(std::string const &str, size_t *pos = 0, int base = 10):

  A conversion as described with stoi is performed, but now to a value of type long long.

• unsigned long stoul(std::string const &str, size_t *pos = 0, int base = 10):

  A conversion as described with stoi (not allowing an initial + or - character) is performed, but now to a value of type unsigned long.
• unsigned long long stoull(std::string const &str, size_t *pos = 0, int base = 10):
  A conversion as described with stoul is performed, but now to a value of type
  unsigned long long.

• std::string to_string(Type value):
  Type can be of the types int, long, long long, unsigned, unsigned long, unsigned long long, float, double, or long double. The value of the argument is converted to a textual representation, which is returned as a std::string value.

• std::string to_wstring(Type value):
  The conversion as described at to_string is performed, returning a std::wstring.
Chapter 6

The IO-stream Library

Extending the standard stream (FILE) approach, well known from the C programming language, C++ offers an input/output (I/O) library based on class concepts.

All C++ I/O facilities are defined in the namespace std. The std:: prefix is omitted below, except for situations where this would result in ambiguities.

Earlier (in chapter 3) we’ve seen several examples of the use of the C++ I/O library, in particular showing insertion operator (<<) and the extraction operator (>>). In this chapter we’ll cover I/O in more detail.

The discussion of input and output facilities provided by the C++ programming language heavily uses the class concept and the notion of member functions. Although class construction has not yet been covered (for that see chapter 7) and although inheritance is not covered formally before chapter 13, it is quite possible to discuss I/O facilities long before the technical background of class construction has been covered.

Most C++ I/O classes have names starting with basic_ (like basic_ios). However, these basic_ names are not regularly found in C++ programs, as most classes are also defined using typedef definitions like:

```cpp
typedef basic_ios<char> ios;
```

Since C++ supports various kinds of character types (e.g., char, wchar_t), I/O facilities were developed using the template mechanism allowing for easy conversions to character types other than the traditional char type. As elaborated in chapter 21, this also allows the construction of generic software, that could thereupon be used for any particular type representing characters. So, analogously to the above typedef there exists a

```cpp
typedef basic_ios<wchar_t> wios;
```

This type definition can be used for the wchar_t type. Because of the existence of these type definitions, the basic_prefix was omitted from the C++ Annotations without loss of continuity. The C++ Annotations primarily focus on the standard 8-bits char type.

Iostream objects cannot be declared using standard forward declarations, like:

```cpp
class std::ostream; // now erroneous
```
Instead, to declare iostream classes the `<iosfwd>` header file should be included:

```
#include <iosfwd> // correct way to declare iostream classes
```

Using C++ I/O offers the additional advantage of type safety. Objects (or plain values) are inserted into streams. Compare this to the situation commonly encountered in C where the `printf` function is used to indicate by a format string what kind of value to expect where. Compared to this latter situation C++'s iostream approach immediately uses the objects where their values should appear, as in

```
cout << "There were " << nMaidens << " virgins present\n";
```

The compiler notices the type of the `nMaidens` variable, inserting its proper value at the appropriate place in the sentence inserted into the `cout` iostream.

Compare this to the situation encountered in C. Although C compilers are getting smarter and smarter, and although a well-designed C compiler may warn you for a mismatch between a format specifier and the type of a variable encountered in the corresponding position of the argument list of a `printf` statement, it can’t do much more than warn you. The type safety seen in C++ prevents you from making type mismatches, as there are no types to match.

Apart from this, iostreams offer more or less the same set of possibilities as the standard FILE-based I/O used in C: files can be opened, closed, positioned, read, written, etc. In C++ the basic FILE structure, as used in C, is still available. But C++ adds to this I/O based on classes, resulting in type safety, extensibility, and a clean design.

In the ANSI/ISO standard the intent was to create architecture independent I/O. Previous implementations of the iostreams library did not always comply with the standard, resulting in many extensions to the standard. The I/O sections of previously developed software may have to be partially rewritten. This is tough for those who are now forced to modify old software, but every feature and extension that was once available can be rebuilt easily using ANSI/ISO standard conforming I/O. Not all of these reimplementations can be covered in this chapter, as many reimplementations rely on inheritance and polymorphism, which topics are formally covered by chapters 13 and 14. Selected reimplementations are provided in chapter 24, and in this chapter references to particular sections in other chapters are given where appropriate. This chapter is organized as follows (see also Figure 6.1):

- The class `ios_base` is the foundation upon which the iostreams I/O library was built. It defines the core of all I/O operations and offers, among other things, facilities for inspecting the state of I/O streams and for output formatting.

- The class `ios` was directly derived from `ios_base`. Every class of the I/O library doing input or output is itself derived from this `ios` class, and so inherits its (and, by implication, `ios_base`'s) capabilities. The reader is urged to keep this in mind while reading this chapter. The concept of inheritance is not discussed here, but rather in chapter 13.

The class `ios` is important in that it implements the communication with a buffer that is used by streams. This buffer is a `streambuf` object which is responsible for the actual I/O to/from the underlying device. Consequently iostream objects do not perform I/O operations themselves, but leave these to the (stream)buffer objects with which they are associated.

- Next, basic C++ output facilities are discussed. The basic class used for output operations is `ostream`, defining the insertion operator as well as other facilities writing information to streams. Apart from inserting information into files it is possible to insert information into memory buffers, for which the `ostringstream` class is available. Formatting output is to a
Figure 6.1: Central I/O Classes
great extent possible using the facilities defined in the `ios` class, but it is also possible to *insert formatting commands* directly into streams using *manipulators*. This aspect of C++ output is discussed as well.

- Basic C++ input facilities are implemented by the `iostream` class. This class defines the extraction operator and related input facilities. Comparably to inserting information into memory buffers (using `ostringstream`) a class `istringstream` is available to extract information from memory buffers.

- Finally, several advanced I/O-related topics are discussed. E.g., reading and writing from the same stream and mixing C and C++ I/O using `filebuf` objects. Other I/O related topics are covered elsewhere in the C++ Annotations, e.g., in chapter 24.

Stream objects have a limited but important role: they are the interface between, on the one hand, the objects to be input or output and, on the other hand, the `streambuf`, which is responsible for the actual input and output to the device accessed by a `streambuf` object.

This approach allows us to construct a new kind of `streambuf` for a new kind of device, and use that `streambuf` in combination with the ‘good old’ `iostream` and `ostream`-class facilities. It is important to understand the distinction between the formatting roles of `iostream` objects and the buffering interface to an external device as implemented in a `streambuf` object. Interfacing to new devices (like sockets or file descriptors) requires the construction of a new kind of `streambuf`, rather than a new kind of `iostream` or `ostream` object. A *wrapper class* may be constructed around the `iostream` or `ostream` classes, though, to ease the access to a special device. This is how the `stringstream` classes were constructed.

### 6.1 Special header files

Several `iostream` related header files are available. Depending on the situation at hand, the following header files should be used:

- `iosfwd`: sources should include this header file if only a declaration of the stream classes is required. For example, if a function defines a reference parameter to an `ostream` then the compiler does not need to know exactly what an `ostream` is. When declaring such a function the `ostream` class merely needs to be be declared. One cannot use

```cpp
  class std::ostream; // erroneous declaration
  void someFunction(std::ostream &str);
```

but, instead, one should use:

```cpp
  #include <iosfwd> // correctly declares class ostream
  void someFunction(std::ostream &str);
```

- `<ios>`: sources should include this header file when using types and facilities (like `ios::off_type`, see below) defined in the `ios` class.

- `<streambuf>`: sources should include this header file when using `streambuf` or `filebuf` classes. See sections 14.8 and 14.8.2.

- `<istream>`: sources should include this preprocessor directive when using the class `istream` or when using classes that do both input and output. See section 6.5.1.
6.2 The foundation: the class ‘ios_base’

The class std::ios_base forms the foundation of all I/O operations, and defines, among other things, facilities for inspecting the state of I/O streams and most output formatting facilities. Every stream class of the I/O library is, through the class ios, derived from this class, and inherits its capabilities. As ios_base is the foundation on which all C++ I/O was built, we introduce it here as the first class of the C++ I/O library.

Note that, as in C, I/O in C++ is not part of the language (although it is part of the ANSI/ISO standard on C++). Although it is technically possible to ignore all predefined I/O facilities, nobody does so, and the I/O library therefore represents a de facto I/O standard for C++. Also note that, as mentioned before, the iostream classes themselves are not responsible for the eventual I/O, but delegate this to an auxiliary class: the class streambuf or its derivatives.

It is neither possible nor required to construct an ios_base object directly. Its construction is always a side-effect of constructing an object further down the class hierarchy, like std::ios. Ios is the next class down the iostream hierarchy (see figure 6.1). Since all stream classes in turn inherit from ios, and thus also from ios_base, the distinction between ios_base and ios is in practice not important. Therefore, facilities actually provided by ios_base will be discussed as facilities provided by ios. The reader who is interested in the true class in which a particular facility is defined should consult the relevant header files (e.g., ios_base.h and basic_ios.h).

6.3 Interfacing ‘streambuf’ objects: the class ‘ios’

The std::ios class is derived directly from ios_base, and it defines de facto the foundation for all stream classes of the C++ I/O library.

Although it is possible to construct an ios object directly, this is seldom done. The purpose of the class ios is to provide the facilities of the class basic_ios, and to add several new facilities, all related to the streambuf object which is managed by objects of the class ios.

All other stream classes are either directly or indirectly derived from ios. This implies, as explained in chapter 13, that all facilities of the classes ios and ios_base are also available to other stream classes. Before discussing these additional stream classes, the features offered by the class ios (and by implication: by ios_base) are now introduced.

In some cases it may be required to include ios explicitly. An example is the situations where the formatting flags themselves (cf. section 6.3.2.2) are referred to in source code.
The class `ios` offers several member functions, most of which are related to formatting. Other frequently used member functions are:

- `std::streambuf *ios::rdbuf()`:
  A pointer to the `streambuf` object forming the interface between the `ios` object and the device with which the `ios` object communicates is returned. See sections 14.8 and 24.1.2 for more information about the class `streambuf`.

- `std::streambuf *ios::rdbuf(std::streambuf *new)`:
  The current `ios` object is associated with another `streambuf` object. A pointer to the `ios` object's original `streambuf` object is returned. The object to which this pointer points is not destroyed when the stream object goes out of scope, but is owned by the caller of `rdbuf`.

- `std::ostream *ios::tie()`:
  A pointer to the `ostream` object that is currently tied to the `ios` object is returned (see the next member). The return value 0 indicates that currently no `ostream` object is tied to the `ios` object. See section 6.5.5 for details.

- `std::ostream *ios::tie(std::ostream *outs)`:
  The `ostream` object is tied to current `ios` object. This means that the `ostream` object is flushed every time before an input or output action is performed by the current `ios` object. A pointer to the `ios` object's original `ostream` object is returned. To break the tie, pass the argument 0. See section 6.5.5 for an example.

### 6.3.1 Condition states

Operations on streams may fail for various reasons. Whenever an operation fails, further operations on the stream are suspended. It is possible to inspect, set and possibly clear the condition state of streams, allowing a program to repair the problem rather than having to abort. The members that are available for interrogating or manipulating the stream's state are described in the current section.

Conditions are represented by the following condition flags:

- `ios::badbit`:
  if this flag has been raised an illegal operation has been requested at the level of the `streambuf` object to which the stream interfaces. See the member functions below for some examples.

- `ios::eofbit`:
  if this flag has been raised, the `ios` object has sensed end of file.

- `ios::failbit`:
  if this flag has been raised, an operation performed by the stream object has failed (like an attempt to extract an `int` when no numeric characters are available on input). In this case the stream itself could not perform the operation that was requested of it.
6.3. INTERFACING ‘STREAMBUF’ OBJECTS: THE CLASS ‘IOS’

- **ios::goodbit**:  
  this flag is raised when none of the other three condition flags were raised.

Several condition member functions are available to manipulate or determine the states of `ios` objects. Originally they returned `int` values, but their current return type is `bool`:

- **bool ios::bad()**:  
  the value `true` is returned when the stream's `badbit` has been set and `false` otherwise. If `true` is returned it indicates that an illegal operation has been requested at the level of the `streambuf` object to which the stream interfaces. What does this mean? It indicates that the `streambuf` itself is behaving unexpectedly. Consider the following example:
  ```cpp
  std::ostream error(0);
  ```
  Here an `ostream` object is constructed _without_ providing it with a working `streambuf` object. Since this ‘streambuf’ will never operate properly, its `badbit` flag is raised from the very beginning: `error.bad()` returns `true`.

- **bool ios::eof()**:  
  the value `true` is returned when end of file (`EOF`) has been sensed (i.e., the `eofbit` flag has been set) and `false` otherwise. Assume we're reading lines line-by-line from `cin`, but the last line is not terminated by a final `\n` character. In that case `std::getline` attempting to read the `\n` delimiter hits end-of-file first. This raises the `eofbit` flag and `cin.eof()` returns `true`. For example, assume `std::string str` and `main` executing the statements:
  ```cpp
  getline(cin, str);
  cout << cin.eof();
  ```
  Then
  ```
  echo "hello world" | program
  ```
  prints the value 0 (no EOF sensed). But after
  ```
  echo -n "hello world" | program
  ```
  the value 1 (EOF sensed) is printed.

- **bool ios::fail()**:  
  the value `true` is returned when `bad` returns `true` or when the `failbit` flag was set. The value `false` is returned otherwise. In the above example, `cin.fail()` returns `false`, whether we terminate the final line with a delimiter or not (as we’ve read a line). However, executing _another_ `getline` results in raising the `failbit` flag, causing `cin::fail()` to return `true`. In general: `fail` returns `true` if the requested stream operation failed. A simple example showing this consists of an attempt to extract an `int` when the input stream contains the text `hello world`. The value not `fail()` is returned by the `bool` interpretation of a stream object (see below).

- **ios::good()**:  
  the value of the `goodbit` flag is returned. It equals `true` when none of the other condition flags (`badbit, eofbit, failbit`) was raised. Consider the following little program:
  ```cpp
  #include <iostream>
  ```
#include <string>

using namespace std;

void state()
{
    cout << "\n"
         "Bad: " << cin.bad() << " 
         "Fail: " << cin.fail() << " 
         "Eof: " << cin.eof() << " 
         "Good: " << cin.good() << '\';
}

int main()
{
    string line;
    int x;

    cin >> x;
    state();

    cin.clear();
    getline(cin, line);
    state();

    getline(cin, line);
    state();
}

When this program processes a file having two lines, containing, respectively, hello and world, while the second line is not terminated by a \n character the following is shown:

Bad: 0 Fail: 1 Eof: 0 Good: 0
Bad: 0 Fail: 0 Eof: 0 Good: 1
Bad: 0 Fail: 0 Eof: 1 Good: 0

Thus, extracting x fails (good returning false). Then, the error state is cleared, and the first line is successfully read (good returning true). Finally the second line is read (incompletely): good returning false, and eof returning true.

- Interpreting streams as bool values:

  streams may be used in expressions expecting logical values. Some examples are:

  if (cin)    // cin itself interpreted as bool
  if (cin >> x) // cin interpreted as bool after an extraction
  if (getline(cin, str)) // getline returning cin

  When interpreting a stream as a logical value, it is actually ‘not fail()’ that is interpreted. The above examples may therefore be rewritten as:

  if (!cin.fail())
  if (!cin >> x).fail())
  if (!getline(cin, str).fail())

  The former incantation, however, is used almost exclusively.
The following members are available to manage error states:

- **void ios::clear():**
  
  When an error condition has occurred, and the condition can be repaired, then `clear` can be used to clear the error state of the file. An overloaded version exists accepting state flags, that are set after first clearing the current set of flags: `clear(int state)`. Its return type is `void`

- **ios::iostate ios::rdstate():**
  
  The current set of flags that are set for an `ios` object are returned (as an `int`). To test for a particular flag, use the bitwise and operator:
  ```cpp
  if (!(iosObject.rdstate() & ios::failbit))
  {
    // last operation didn’t fail
  }
  ```
  
  Note that this test cannot be performed for the `goodbit` flag as its value equals zero. To test for ‘good’ use a construction like:
  ```cpp
  if (iosObject.rdstate() == ios::goodbit)
  {
    // state is ‘good’
  }
  ```

- **void ios::setstate(ios::iostate state):**
  
  A stream may be assigned a certain set of states using `setstate`. Its return type is `void`. E.g.,
  ```cpp
  cin.setstate(ios::failbit);  // set state to ‘fail’
  ```
  
  To set multiple flags in one `setstate()` call use the `bitor` operator:
  ```cpp
  cin.setstate(ios::failbit | ios::eofbit)
  ```
  
  The member `clear` is a shortcut to clear all error flags. Of course, clearing the flags doesn’t automatically mean the error condition has been cleared too. The strategy should be:
  
  - An error condition is detected,
  - The error is repaired
  - The member `clear` is called.

C++ supports an *exception* mechanism to handle exceptional situations. According to the ANSI/ISO standard, exceptions can be used with stream objects. Exceptions are covered in chapter 10. Using exceptions with stream objects is covered in section 10.7.

### 6.3.2 Formatting output and input

The way information is written to streams (or, occasionally, read from streams) is controlled by *formatting flags*.

Formatting is used when it is necessary to, e.g., set the width of an output field or input buffer and to determine the form (e.g., the `radix`) in which values are displayed. Most formatting features belong to the realm of the `ios` class. Formatting is controlled by flags, defined by the `ios` class. These flags may be manipulated in two ways: using specialized member functions or using *manipulators*, which
are directly inserted into or extracted from streams. There is no special reason for using either method; usually both methods are possible. In the following overview the various member functions are first introduced. Following this the flags and manipulators themselves are covered. Examples are provided showing how the flags can be manipulated and what their effects are.

Many manipulators are parameterless and are available once a stream header file (e.g., `iostream`) has been included. Some manipulators require arguments. To use the latter manipulators the header file `iomanip` must be included.

### Format modifying member functions

Several **member functions** are available manipulating the I/O formatting flags. Instead of using the members listed below **manipulators** are often available that may directly be inserted into or extracted from streams. The available members are listed in alphabetical order, but the most important ones in practice are `setf`, `unsetf` and `width`.

- **`ios &ios::copyfmt(ios &obj):`**
  
  all format flags of `obj` are copied to the current `ios` object. The current `ios` object is returned.

- **`ios::fill() const`**
  
  the current padding character is returned. By default, this is the blank space.

- **`ios::fill(char padding)`**
  
  the padding character is redefined, the previous padding character is returned. Instead of using this member function the `setfill` manipulator may be inserted directly into an `ostream`. Example:

  ```cpp
  cout.fill('0'); // use '0' as padding char
  cout << setfill('+'); // use '+' as padding char
  ```

- **`ios::fmtflags ios::flags() const`**
  
  the current set of flags controlling the format state of the stream for which the member function is called is returned. To inspect whether a particular flag was set, use the `bit_and` operator. Example:

  ```cpp
  if (cout.flags() & ios::hex)
    cout << "Integral values are printed as hex numbers\n"
  ```

- **`ios::fmtflags ios::flags(ios::fmtflags flagset)`**
  
  the previous set of flags are returned and the new set of flags are defined by `flagset`. Multiple flags are specified using the `bitor` operator. Example:

  ```cpp
  // change the representation to hexadecimal
  cout.flags(ios::hex | cout.flags() & ~ios::dec);
  ```

- **`int ios::precision() const`**
  
  the number of significant digits used when outputting floating point values is returned (default: 6).
6.3. INTERFACING ‘STREAMBUF’ OBJECTS: THE CLASS ‘IOS’

- **int ios::precision(int signif):**
  
  the number of significant digits to use when outputting real values is set to `signif`. The previously used number of significant digits is returned. If the number of required digits exceeds `signif` then the number is displayed in ‘scientific’ notation (cf. section 6.3.2.2). Manipulator: `setprecision`. Example:
  
  ```cpp
  cout.precision(3); // 3 digits precision
  cout << setprecision(3); // same, using the manipulator
  cout << 1.23 << " " << 12.3 << " " << 123.12 << " " << 1234.3 << '\n';
  // displays: 1.23 12.3 123 1.23e+03
  ```

- **ios::fmtflags ios::setf(ios::fmtflags flags):**

  sets one or more formatting flags (use the `bitor` operator to combine multiple flags). Already set flags are not affected. The previous set of flags is returned. Instead of using this member function the manipulator `setiosflags` may be used. Examples are provided in the next section (6.3.2.2).

- **ios::fmtflags ios::setf(ios::fmtflags flags, ios::fmtflags mask):**

  clears all flags mentioned in `mask` and sets the flags specified in `flags`. The previous set of flags is returned. Some examples are (but see the next section (6.3.2.2) for a more thorough discussion):
  
  ```cpp
  // left-adjust information in wide fields
  cout.setf(ios::left, ios::adjustfield);
  // display integral values as hexadecimal numbers
  cout.setf(ios::hex, ios::basefield);
  // display floating point values in scientific notation
  cout.setf(ios::scientific, ios::floatfield);
  ```

- **ios::fmtflags ios::unsetf(fmtflags flags):**

  the specified formatting flags are cleared (leaving the remaining flags unaltered) and returns the previous set of flags. A request to unset an active default flag (e.g., `cout.unsetf(ios::dec)`) is ignored. Instead of this member function the manipulator `resetiosflags` may also be used. Example:
  
  ```cpp
  cout << 12.24; // displays 12.24
  cout << setf(ios::fixed);
  cout << 12.24; // displays 12.240000
  cout.unsetf(ios::fixed); // undo a previous ios::fixed setting.
  cout << 12.24; // displays 12.24
  cout << resetiosflags(ios::fixed); // using manipulator rather
  // than unsetf
  ```

- **int ios::width() const:**

  the currently active output field width to use on the next insertion is returned. The default value is 0, meaning ‘as many characters as needed to write the value’.

- **int ios::width(int nchars):**

  the field width of the next insertion operation is set to `nchars`, returning the previously used field width. This setting is not persistent. It is reset to 0 after every insertion operation. Manipulator: `std::setw(int)`. Example:
  
  ```cpp
  cout.width(5);
  cout << 12; // using 5 chars field width
  cout << setw(12) << "hello"; // using 12 chars field width
  ```
6.3.2.2 Formatting flags

Most formatting flags are related to outputting information. Information can be written to output streams in basically two ways: using binary output information is written directly to an output stream, without converting it first to some human-readable format and using formatted output by which values stored in the computer’s memory are converted to human-readable text first. Formatting flags are used to define the way this conversion takes place. In this section all formatting flags are covered. Formatting flags may be (un)set using member functions, but often manipulators having the same effect may also be used. For each of the flags it is shown how they can be controlled by a member function or -if available- a manipulator.

To display information in wide fields:

- **ios::internal:**
  
  to add fill characters (blanks by default) between the minus sign of negative numbers and the value itself. Other values and data types are right-adjusted. Manipulator: std::internal. Example:

  ```
  cout.setf(ios::internal, ios::adjustfield);
  cout << internal; // same, using the manipulator
  cout << '\n' << setw(5) << -5 << '\n'; // displays '-' 5
  ```

- **ios::left:**
  
  to left-adjust values in fields that are wider than needed to display the values. Manipulator: std::left. Example:

  ```
  cout.setf(ios::left, ios::adjustfield);
  cout << left; // same, using the manipulator
  cout << '\n' << setw(5) << "hi" << '\n"; // displays 'hi'
  ```

- **ios::right:**
  
  to right-adjust values in fields that are wider than needed to display the values. Manipulator: std::right. This is the default. Example:

  ```
  cout.setf(ios::right, ios::adjustfield);
  cout << right; // same, using the manipulator
  cout << '\n' << setw(5) << "hi" << '\n"; // displays ' hi'
  ```

Using various number representations:

- **ios::dec:**
  
  to display integral values as decimal numbers. Manipulator: std::dec. This is the default. Example:

  ```
  cout.setf(ios::dec, ios::basefield);
  cout << dec; // same, using the manipulator
  cout << 0x10; // displays 16
  ```
6.3. INTERFACING ‘STREAMBUF’ OBJECTS: THE CLASS ‘IOS’

- **ios::hex:**
  
  To display integral values as hexadecimal numbers. **Manipulator:** std::hex. Example:
  
  ```cpp
cout.setf(ios::hex, ios::basefield);
cout << hex; // same, using the manipulator
cout << 16;   // displays 10
```

- **ios::oct:**
  
  To display integral values as octal numbers. **Manipulator:** std::oct. Example:
  
  ```cpp
cout.setf(ios::oct, ios::basefield);
cout << oct;  // same, using the manipulator
cout << 16;   // displays 20
```

- **std::setbase(int radix):**
  
  This is a manipulator that can be used to change the number representation to decimal, hexadecimal or octal. Example:
  
  ```cpp
cout << setbase(8); // octal numbers, use 10 for
cout << 16;       // decimal, 16 for hexadecimal
```

**Fine-tuning displaying values:**

- **ios::boolalpha:**
  
  Logical values may be displayed as text using the text ‘true’ for the true logical value, and ‘false’ for the false logical value using boolalpha. By default this flag is not set. **Complementary flag:** ios::noboolalpha. **Manipulators:** std::boolalpha and std::noboolalpha. Example:
  
  ```cpp
cout.setf(ios::boolalpha);
cout << boolalpha; // same, using the manipulator
cout << (1 == 1);   // displays true
```

- **ios::showbase:**
  
  To display the numeric base of integral values. With hexadecimal values the 0x prefix is used, with octal values the prefix 0. For the (default) decimal value no particular prefix is used. **Complementary flag:** ios::noshowbase. **Manipulators:** std::showbase and std::noshowbase. Example:
  
  ```cpp
cout.setf(ios::showbase);
cout << showbase;  // same, using the manipulator
cout << hex << 16; // displays 0x10
```

- **ios::showpos:**
  
  To display the + sign with positive decimal (only) values. **Complementary flag:** ios::noshowpos. **Manipulators:** std::showpos and std::noshowpos. Example:
  
  ```cpp
cout.setf(ios::showpos);
cout << showpos;   // same, using the manipulator
cout << 16;        // displays +16
cout.unsetf(ios::showpos); // Undo showpos
cout << 16;        // displays 16
```
• `ios::uppercase`:

  to display letters in hexadecimal values using capital letters. Complementary flag: `ios::nouppercase`. Manipulators: `std::uppercase` and `std::nouppercase`. By default lower case letters are used. Example:

  ```cpp
  cout.setf(ios::uppercase);
  cout << uppercase; // same, using the manipulator
  cout << hex << showbase << 3735928559; // displays 0XDEADBEEF
  ```

Displaying floating point numbers

• `ios::fixed`:

  to display real values using a fixed decimal point (e.g., 12.25 rather than 1.225e+01), the `fixed` formatting flag is used. It can be used to set a fixed number of digits behind the decimal point. Manipulator: `fixed`. Example:

  ```cpp
  cout.setf(ios::fixed, ios::floatfield);
  cout.precision(3); // 3 digits behind the .
  // Alternatively:
  cout << setiosflags(ios::fixed) << setprecision(3);
  cout << 3.0 << " " << 3.01 << " " << 3.001 << ‘\n’;
  << 3.0004 << " " << 3.0005 << " " << 3.0006 << ‘\n’
  // Results in:
  // 3.000 3.010 3.001
  // 3.000 3.001 3.001
  ```

  The example shows that 3.0005 is rounded away from zero, becoming 3.001 (likewise -3.0005 becomes -3.001). First setting precision and then `fixed` has the same effect.

• `ios::scientific`:

  to display real values in **scientific notation** (e.g., 1.24e+03). Manipulator: `std::scientific`. Example:

  ```cpp
  cout.setf(ios::scientific, ios::floatfield);
  cout << scientific; // same, using the manipulator
  cout << 12.25; // displays 1.22500e+01
  ```

• `ios::showpoint`:

  to display a trailing decimal point and trailing decimal zeros when real numbers are displayed. Complementary flag: `ios::noshowpoint`. Manipulators: `std::showpoint`, `std::noshowpoint`. Example:

  ```cpp
  cout << fixed << setprecision(3); // 3 digits behind .
  cout.setf(ios::showpoint); // set the flag
  cout << showpoint; // same, using the manipulator
  cout << 16.0 << ", " << 16.1 << ", " << 16;
  // displays: 16.000, 16.100, 16
  ```
Note that the final 16 is an integral rather than a floating point number, so it has no decimal point. So showpoint has no effect. If ios::showpoint is not active trailing zeros are discarded. If the fraction is zero the decimal point is discarded as well. Example:

```cpp
cout.unsetf(ios::fixed, ios::showpoint); // unset the flags
cout << 16.0 << ", " << 16.1;
// displays: 16, 16.1
```

**Handling white space and flushing streams**

- **std::endl:**
  - manipulator inserting a newline character and flushing the stream. Often flushing the stream is not required and doing so would needlessly slow down I/O processing. Consequently, using \endl should be avoided (in favor of inserting ‘\n’) unless flushing the stream is explicitly intended. Note that streams are automatically flushed when the program terminates or when a stream is ‘tied’ to another stream (cf. tie in section 6.3). Example:

  ```cpp
cout << "hello" << endl; // prefer: << '\n';
```

- **std::ends:**
  - manipulator inserting a 0-byte into a stream. It is usually used in combination with memory-streams (cf. section 6.4.3).

- **std::flush:**
  - a stream may be flushed using this member. Often flushing the stream is not required and doing so would needlessly slow down I/O processing. Consequently, using flush should be avoided unless it is explicitly required to do so. Note that streams are automatically flushed when the program terminates or when a stream is ‘tied’ to another stream (cf. tie in section 6.3). Example:

  ```cpp
cout << "hello" << flush; // avoid if possible.
```

- **ios::skipws:**
  - leading white space characters (blanks, tabs, newlines, etc.) are skipped when a value is extracted from a stream. This is the default. If the flag is not set, leading white space characters are not skipped. Manipulator: std::skipws. Example:

  ```cpp
cin.setf(ios::skipws); // to unset, use
  // cin.unsetf(ios::skipws)

  cin >> skipws; // same, using the manipulator
  int value;
  cin >> value; // skips initial blanks
```

- **ios::unitbuf:**
  - the stream for which this flag is set flushes its buffer after every output operation. Often flushing a stream is not required and doing so would needlessly slow down I/O processing. Consequently, setting unitbuf should be avoided unless flushing the stream is explicitly intended. Note that streams are automatically flushed when the
program terminates or when a stream is ‘tied’ to another stream (cf. tie in section 6.3). Complementary flag: ios::nouinitbuf. Manipulators: std::unitbuf, std::nounitbuf. Example:

```cpp
cout.setf(ios::unitbuf);
cout << unitbuf;  // same, using the manipulator
cout.write("xyz", 3);  // flush follows write.
```

- std::ws: manipulator removing all white space characters (blanks, tabs, newlines, etc.) at the current file position. White space are removed if present even if the flag ios::noskipws has been set. Example (assume the input contains 4 blank characters followed by the character ‘X’):

```cpp
cin >> ws;  // skip white space
cin.get();  // returns ‘X’
```

### 6.4 Output

In C++ output is primarily based on the std::ostream class. The ostream class defines the basic operators and members inserting information into streams: the insertion operator (<<), and special members like write writing unformatted information to streams.

The class ostream acts as base class for several other classes, all offering the functionality of the ostream class, but adding their own specialties. In the upcoming sections the following classes are discussed:

- The class ostream, offering the basic output facilities;
- The class ofstream, allowing us to write files (comparable to C’s fopen(filename, "w"));
- The class ostringstream, allowing us to write information to memory (comparable to C’s sprintf function).

#### 6.4.1 Basic output: the class ‘ostream’

The class ostream defines basic output facilities. The cout, clog and cerr objects are all ostream objects. All facilities related to output as defined by the ios class are also available in the ostream class.

We may define ostream objects using the following ostream constructor:

```cpp
std::ostream object(std::streambuf *sb):
```

this constructor creates an ostream object which is a wrapper around an existing std::streambuf object. It isn’t possible to define a plain ostream object (e.g., using std::ostream out;) that can thereupon be used for insertions. When cout or its friends are used, we are actually using a predefined ostream object that has already been defined for us and interfaces to the standard output stream using a (also predefined) streambuf object handling the actual interfacing.

It is, however, possible to define an ostream object passing it a 0-pointer. Such an object cannot be used for insertions (i.e., it raises its ios::bad flag when something
is inserted into it), but it may be given a streambuf later. Thus it may be preliminary constructed, suspending its use until an appropriate streambuf becomes available (see also section 13.6).

To define the ostream class in C++ sources, the <ostream> header file must be included. To use the predefined ostream objects (std::cin, std::cout etc.) the <iostream> header file must be included.

### 6.4.1.1 Writing to ‘ostream’ objects

The class ostream supports both formatted and binary output.

The insertion operator (<<) is used to insert values in a type safe way into ostream objects. This is called formatted output, as binary values which are stored in the computer's memory are converted to human-readable ASCII characters according to certain formatting rules.

The insertion operator points to the ostream object to receive the information. The normal associativity of << remains unaltered, so when a statement like

```cpp
    cout << "hello " << "world";
```

is encountered, the leftmost two operands are evaluated first (cout << "hello "), and an ostream & object, which is actually the same cout object, is returned. Now, the statement is reduced to

```cpp
    cout << "world";
```

and the second string is inserted into cout.

The << operator has a lot of (overloaded) variants, so many types of variables can be inserted into ostream objects. There is an overloaded <<-operator expecting an int, a double, a pointer, etc. etc.. Each operator returns the ostream object into which the information so far has been inserted, and can thus immediately be followed by the next insertion.

Streams lack facilities for formatted output like C's printf and vprintf functions. Although it is not difficult to implement these facilities in the world of streams, printf-like functionality is hardly ever required in C++ programs. Furthermore, as it is potentially type-unsafe, it might be better to avoid this functionality completely.

When binary files must be written, normally no text-formatting is used or required: an int value should be written as a series of raw bytes, not as a series of ASCII numeric characters 0 to 9. The following member functions of ostream objects may be used to write ‘binary files’:

- **ostream& put(char c):**
  
  to write a single character to the output stream. Since a character is a byte, this member function could also be used for writing a single character to a text-file.

- **ostream& write(char const *buffer, int length):**
  
  to write at most length bytes, stored in the char const *buffer to the ostream object. Bytes are written as they are stored in the buffer, no formatting is done whatsoever. Note that the first argument is a char const *: a type cast is required to write any other type. For example, to write an int as an unformatted series of byte-values use:

```cpp
    int x;
```
out.write(reinterpret_cast<char const *>(&x), sizeof(int));

The bytes written by the above write call are written to the ostream in an order depending on the endian-ness of the underlying hardware. Big-endian computers write the most significant byte(s) of multi-byte values first, little-endian computers first write the least significant byte(s).

6.4.1.2 ‘ostream’ positioning

Although not every ostream object supports repositioning, they usually do. This means that it is possible to rewrite a section of the stream which was written earlier. Repositioning is frequently used in database applications where it must be possible to access the information in the database at random.

The current position can be obtained and modified using the following members:

- `ios::pos_type tellp()`:
  the current (absolute) position in the file where the next write-operation to the stream will take place is returned.

- `ostream &seekp(ios::off_type step, ios::seekdir org)`:
  modifies a stream’s actual position. The function expects an off_type step representing the number of bytes the current stream position is moved with respect to org. The step value may be negative, zero or positive. The origin of the step, org is a value in the ios::seekdir enumeration. Its values are:
  - `ios::beg`:
    the stepsizes is computed relative to the beginning of the stream. This value is used by default.
  - `ios::cur`:
    the stepsizes is computed relative to the current position of the stream (as returned by tellp).
  - `ios::end`:
    the stepsizes is interpreted relative to the current end position of the stream.

It is OK to seek or write beyond the last file position. Writing bytes to a location beyond EOF will pad the intermediate bytes with 0-valued bytes: null-bytes. Seeking before `ios::beg` raises the `ios::fail` flag.

6.4.1.3 ‘ostream’ flushing

Unless the `ios::unitbuf` flag has been set, information written to an ostream object is not immediately written to the physical stream. Rather, an internal buffer is filled during the write-operations, and when full it is flushed.

The stream’s internal buffer can be flushed under program control:

- `ostream& flush()`:
  any buffered information stored internally by the ostream object is flushed to the device to which the ostream object interfaces. A stream is flushed automatically when:
    - the object ceases to exist;
6.4. Output

- the `endl` or `flush` manipulators (see section 6.3.2.2) are inserted into an `ostream` object;
- a stream supporting the close-operation is explicitly closed (e.g., a `std::ofstream` object, cf. section 6.4.2).

6.4.2 Output to files: the class `ofstream`

The `std::ofstream` class is derived from the `ostream` class: it has the same capabilities as the `ostream` class, but can be used to access files or create files for writing.

In order to use the `ofstream` class in C++ sources, the `<fstream>` header file must be included. Including `fstream` does not automatically make available the standard streams `cin`, `cout` and `cerr`. Include `iostream` to declare these standard streams.

The following constructors are available for `ofstream` objects:

- `ofstream object`:
  
  this is the basic constructor. It defines an `ofstream` object which may be associated with an actual file later, using its `open()` member (see below).

- `ofstream object(char const *name, ios::openmode mode = ios::out)`:
  
  this constructor defines an `ofstream` object and associates it immediately with the file named `name` using output mode `mode`. Section 6.4.2.1 provides an overview of available output modes. Example:
  ```cpp
  ofstream out("/tmp/scratch");
  ```

  It is not possible to open an `ofstream` using a file descriptor. The reason for this is (apparently) that file descriptors are not universally available over different operating systems. Fortunately, file descriptors can be used (indirectly) with a `std::streambuf` object (and in some implementations: with a `std::filebuf` object, which is also a `streambuf`). Streambuf objects are discussed in section 14.8, filebuf objects are discussed in section 14.8.2.

  Instead of directly associating an `ofstream` object with a file, the object can be constructed first, and opened later:

  - `void open(char const *name, ios::openmode mode = ios::out)`: 
    
    associates an `ofstream` object with an actual file. If the `ios::fail` flag was set before calling `open` and opening succeeds the flag is cleared. Opening an already open stream fails. To reassociate a stream with another file it must first be closed:
    ```cpp
    ofstream out("/tmp/out");
    out << "hello\n";
    out.close(); // flushes and closes out
    out.open("/tmp/out2");
    out << "world\n";
    ```

  - `void close()`:
    
    closes the `ofstream` object. The function sets the `ios::fail` flag of the closed object. Closing the file flushes any buffered information to the associated file. A file is automatically closed when the associated `ofstream` object ceases to exist.
CHAPTER 6. THE IO-STREAM LIBRARY

```cpp
bool is_open() const

assume a stream was properly constructed, but it has not yet been attached to a file. E.g., the statement ofstream ostr was executed. When we now check its status through good(), a non-zero (i.e., OK) value is returned. The 'good' status here indicates that the stream object has been constructed properly. It doesn’t mean the file is also open. To test whether a stream is actually open, is_open should be called. If it returns true, the stream is open. Example:

```cpp
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    ofstream of;
    cout << "of’s open state: " << boolalpha << of.is_open() << ‘\n’;
    of.open("/dev/null");  // on Unix systems
    cout << "of’s open state: " << of.is_open() << ‘\n’;
}
/*
 Generated output:
of’s open state: false
of’s open state: true
*/
```

6.4.2.1 Modes for opening stream objects

The following file modes or file flags are available when constructing or opening ofstream (or istream, see section 6.5.2) objects. The values are of type ios::openmode. Flags may be combined using the bitor operator.

- **ios::app:**
  reposition the stream to its end before every output command (see also ios::ate below). The file is created if it doesn’t yet exist. When opening a stream in this mode any existing contents of the file are kept.

- **ios::ate:**
  start initially at the end of the file. Note that any existing contents are only kept if some other flag tells the object to do so. For example ofstream out("gone", ios::ate) rewrites the file gone, because the implied ios::out causes the rewriting. If rewriting of an existing file should be prevented, the ios::in mode should be specified too. However, when ios::in is specified the file must already exist. The ate mode only initially positions the file at the end of file position. After that information may be written in the middle of the file using seekp. When the app mode is used information is only written at end of file (effectively ignoring seekp operations).

- **ios::binary:**
  open a file in binary mode (used on systems distinguishing text- and binary files, like MS-Windows).
6.4. OUTPUT

- **ios::in:**
  open the file for reading. The file must exist.

- **ios::out:**
  open the file for writing. Create it if it doesn’t yet exist. If it exists, the file is rewritten.

- **ios::trunc:**
  start initially with an empty file. Any existing contents of the file are lost.

The following combinations of file flags have special meanings:

<table>
<thead>
<tr>
<th>Combination</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>out</td>
</tr>
<tr>
<td>in</td>
<td>out</td>
</tr>
</tbody>
</table>

An interesting subtlety is that the open members of the ifstream, ofstream and fstream classes have a second parameter of type ios::openmode. In contrast to this, the bitor operator returns an int when applied to two enum-values. The question why the bitor operator may nevertheless be used here is answered in a later chapter (cf. section 11.11).

6.4.3 Output to memory: the class ‘ostringstream’

To write information to memory using stream facilities, std::ostringstream objects should be used. As the class ostringstream is derived from the class ostream all ostream’s facilities are available to ostringstream objects as well. To use and define ostringstream objects the header file <sstream> must be included. In addition the class ostringstream offers the following constructors and members:

- **ostringstream ostr(string const &init, ios::openmode mode = ios::out):**
  when specifying openmode as ios::ate, the ostringstream object is initialized by the string init and remaining insertions are appended to the contents of the ostringstream object.

- **ostringstream ostr(ios::openmode mode = ios::out):**
  this constructor can also be used as default constructor. Alternatively it allows, e.g., forced additions at the end of the information stored in the object so far (using ios::app). Example:
  ```
  std::ostringstream out;
  ```

- **std::string str() const:**
  a copy of the string that is stored inside the ostringstream object is returned.

- **void str(std::string const &str):**
  the current object is reinitialized with new initial contents.
The following example illustrates the use of the `ostringstream` class: several values are inserted into the object. Then, the text contained by the `ostringstream` object is stored in a `std::string`, whose length and contents are thereupon printed. Such `ostringstream` objects are most often used for doing ‘type to string’ conversions, like converting `int` values to text. Formatting flags can be used with `ostringstream` as well, as they are part of the `ostream` class.

Here is an example showing an `ostringstream` object being used:

```cpp
#include <iostream>
#include <sstream>

using namespace std;

int main()
{
    ostringstream ostr("hello ", ios::ate);
    cout << ostr.str() << '
';
    ostr.setf(ios::showbase);
    ostr.setf(ios::hex, ios::basefield);
    ostr << 12345;
    cout << ostr.str() << '
';
    ostr << " -- ";
    ostr.unsetf(ios::hex);
    ostr << 12;
    cout << ostr.str() << '
';
    ostr.str("new text");
    cout << ostr.str() << '
';
    ostr.seekp(4, ios::beg);
    ostr << "world";
    cout << ostr.str() << '
';
}
/*
 * Output from this program:
 hello
 hello 0x3039
 hello 0x3039 -- 12
 new text
 new world
 */
```

### 6.5 Input

In C++ input is primarily based on the `std::istream` class. The `istream` class defines the basic operators and members extracting information from streams: the extraction operator (`>>`), and special members like `istream::read` reading unformatted information from streams.
The class `istream` acts as base class for several other classes, all offering the functionality of the `istream` class, but adding their own specialties. In the upcoming sections the following classes are discussed:

- The class `istream`, offering the basic facilities for doing input;
- The class `ifstream`, allowing us to read files (comparable to C's `fopen(filename, "r")`);
- The class `istringstream`, allowing us to read information from text that is not stored on files (streams) but in memory (comparable to C's `sscanf` function).

### 6.5.1 Basic input: the class ‘istream’

The class `istream` defines basic input facilities. The `cin` object, is an `istream` object. All facilities related to input as defined by the `ios` class are also available in the `istream` class.

We may define `istream` objects using the following `istream` constructor:

- `istream object(streambuf *sb)`:

  this constructor can be used to construct a wrapper around an existing `std::streambuf` object. Similarly to `ostream` objects, `istream` objects may be defined by passing it initially a 0-pointer. See section 6.4.1 for a discussion, see also section 13.6, and see chapter 24 for examples.

To define the `istream` class in C++ sources, the `<iostream>` header file must be included. To use the predefined `istream` object `cin`, the `<iostream>` header file must be included.

#### 6.5.1.1 Reading from ‘istream’ objects

The class `istream` supports both formatted and unformatted binary input. The extraction operator (`operator >>`) is used to extract values in a type safe way from `istream` objects. This is called formatted input, whereby human-readable ASCII characters are converted, according to certain formatting rules, to binary values.

The extraction operator points to the objects or variables which receive new values. The normal associativity of `>>` remains unaltered, so when a statement like

```cpp
    cin >> x >> y;
```

is encountered, the leftmost two operands are evaluated first (`cin >> x`), and an `istream &` object, which is actually the same `cin` object, is returned. Now, the statement is reduced to

```cpp
    cin >> y
```

and the `y` variable is extracted from `cin`.

The `>>` operator has many (overloaded) variants and thus many types of variables can be extracted from `istream` objects. There is an overloaded `>>` available for the extraction of an `int`, of a `double`, of a `string`, of an array of characters, possibly to a pointer, etc. etc. String or character array extraction by default first skips all white space characters, and then extracts all consecutive non-white space characters. Once an extraction operator has been processed the `istream` object from
which the information was extracted is returned and it can immediately be used for additional
*istream* operations that appear in the same expression.

Streams lack facilities for formatted input (as used by, e.g., C’s *scanf* and *vscanf* functions). Al-
though it is not difficult to add these facilities to the world of streams, *scanf*-like functionality is
hardly ever required in C++ programs. Furthermore, as it is potentially type-unsafe, it might be
to avoid formatted input completely.

When binary files must be read, the information should normally not be formatted: an *int* value
should be read as a series of unaltered bytes, not as a series of ASCII numeric characters 0 to 9. The
following member functions for reading information from *istream* objects are available:

- **int gcoun**t() const:
  the number of characters read from the input stream by the last unformatted input
  operation is returned.

- **int get()**:
  the next available single character is returned as an unsigned char value using an
  *int* return type. EOF is returned if no more character are available.

- **istream &get(char &ch)**:
  the next single character read from the input stream is store in ch. The member
  function returns the stream itself which may be inspected to determine whether a
  character was obtained or not.

- **istream& get(char *buffer, int len, char delim = ‘\n’)***:
  At most len − 1 characters are read from the input stream into the array starting
  at buffer, which should be at least len bytes long. Reading also stops when the
delimiter delim is encountered. However, the delimiter itself is not removed from
the input stream.

  Having stored the characters into buffer, an 0-valued character is written beyond
  the last character stored into the buffer. The functions *eof* and *fail* (see section
6.3.1) return 0 (false) if the delimiter was encountered before reading len − 1
characters or if the delimiter was not encountered after reading len − 1 characters.
It is OK to specify an 0-valued character delimiter: this way NTB strings may be
read from a (binary) file.

- **istream& getline(char *buffer, int len, char delim = ‘\n’)***:
  this member function operates analogously to the get member function, but *getline*
removes delim from the stream if it is actually encountered. The delimiter itself, if
encountered, is not stored in the buffer. If delim was not found (before reading
len − 1 characters) the fail member function, and possibly also *eof* returns true.

  Realize that the std::string class also offers a function std::getline which is
generally preferred over this getline member function that is described here (see
section 5.2.4).

- **istream& ignore()**:
  one character is skipped from the input stream.

- **istream& ignore(int n)**:
  *n* characters are skipped from the input stream.
6.5. INPUT

- **istream& ignore(int n, int delim):**
  
  at most n characters are skipped but skipping characters stops after having removed delim from the input stream.

- **int peek():**
  
  this function returns the next available input character, but does not actually remove the character from the input stream. **EOF** is returned if no more characters are available.

- **istream& putback(char ch):**
  
  The character ch is ‘pushed back’ into the input stream, to be read again as the next available character. **EOF** is returned if this is not allowed. Normally, it is OK to put back one character. Example:

  ```cpp
  string value;
  cin >> value;
  cin.putback('X');  // displays: X
  cout << static_cast<char>(cin.get());
  ```

- **istream &read(char *buffer, int len):**
  
  At most len bytes are read from the input stream into the buffer. If **EOF** is encountered first, fewer bytes are read, with the member function **eof** returning **true**. This function is commonly used when reading binary files. Section 6.5.2 contains an example in which this member function is used. The member function **gcount()** may be used to determine the number of characters that were retrieved by **read**.

- **istream& readsome(char *buffer, int len):**
  
  at most len bytes are read from the input stream into the buffer. All available characters are read into the buffer, but if **EOF** is encountered, fewer bytes are read, without setting the **ios::eofbit** or **ios::failbit**.

- **istream &unget():**
  
  the last character that was read from the stream is put back.

6.5.1.2 ‘istream’ positioning

Although not every **istream** object supports repositioning, some do. This means that it is possible to read the same section of a stream repeatedly. Repositioning is frequently used in database applications where it must be possible to access the information in the database randomly.

The current position can be obtained and modified using the following members:

- **ios::pos_type tellg():**
  
  the stream’s current (absolute) position where the stream’s next read-operation will take place is returned.

- **istream &seekg(ios::off_type step, ios::seekdir org):**
  
  modifies a stream’s actual position. The function expects an **off_type step** representing the number of bytes the current stream position is moved with respect to org. The **step** value may be negative, zero or positive.
The origin of the step, org is a value in the ios::seekdir enumeration. Its values are:

- ios::beg:
  the stepsize is computed relative to the beginning of the stream. This value is used by default.
- ios::cur:
  the stepsize is computed relative to the current position of the stream (as returned by tellp).
- ios::end:
  the stepsize is interpreted relative to the current end position of the stream.

It is OK to seek beyond the last file position. Seeking before ios::beg raises the ios::failbit flag.

6.5.2 Input from files: the class ‘ifstream’

The std::ifstream class is derived from the istream class: it has the same capabilities as the istream class, but can be used to access files for reading.

In order to use the ifstream class in C++ sources, the <fstream> header file must be included. Including fstream does not automatically make available the standard streams cin, cout and cerr. Include iostream to declare these standard streams.

The following constructors are available for ifstream objects:

- ifstream object:
  this is the basic constructor. It defines an ifstream object which may be associated with an actual file later, using its open() member (see below).

- ifstream object(char const *name, ios::openmode mode = ios::in):
  this constructor can be used to define an ifstream object and associate it immediately with the file named name using input mode mode. Section 6.4.2.1 provides an overview of available input modes. Example:

  ifstream in("/tmp/input");

Instead of directly associating an ifstream object with a file, the object can be constructed first, and opened later.

- void open(char const *name, ios::openmode mode = ios::in):
  associates an ifstream object with an actual file. If the ios::fail flag was set before calling open and opening succeeds the flag is cleared. Opening an already open stream fails. To reassociate a stream with another file it must first be closed:

  ifstream in("/tmp/in");
  in >> variable;
  in.close(); // closes in
  in.open("/tmp/in2");
  in >> anotherVariable;
6.5. INPUT

- **void close()**: closes the ifstream object. The function sets the ios::fail flag of the closed object. Closing the file flushes any buffered information to the associated file. A file is automatically closed when the associated ifstream object ceases to exist.

- **bool is_open() const**: assume a stream was properly constructed, but it has not yet been attached to a file. E.g., the statement ifstream ostr was executed. When we now check its status through good(), a non-zero (i.e., OK) value is returned. The 'good' status here indicates that the stream object has been constructed properly. It doesn't mean the file is also open. To test whether a stream is actually open, is_open should be called. If it returns true, the stream is open. Also see the example in section 6.4.2. The following example illustrates reading from a binary file (see also section 6.5.1.1):

```cpp
#include <fstream>
using namespace std;

int main(int argc, char**argv)
{
    ifstream in(argv[1]);
    double value;

    // reads double in raw, binary form from file.
    in.read(reinterpret_cast<char*>(&value), sizeof(double));
}
```

### 6.5.3 Input from memory: the class ‘istringstream’

To read information from memory using stream facilities, std::istringstream objects should be used. As the class istringstream is derived from the class istream all istream's facilities are available to istringstream objects as well. To use and define istringstream objects the header file <sstream> must be included. In addition the class istringstream offers the following constructors and members:

- **sstream istr(string const &init, ios::openmode mode = ios::in)**: the object is initialized with init’s contents

- **sstream istr(ios::openmode mode = ios::in)** (this constructor is usually used as the default constructor. Example:

  ```cpp
  std::istringstream in;
  ```

- **void str(std::string const &str)**: the current object is reinitialized with new initial contents.

The following example illustrates the use of the istringstream class: several values are extracted from the object. Such istringstream objects are most often used for doing 'string to type' conversions, like converting text to int values (cf. C’s atoi function). Formatting flags can be used with
istreamstreams as well, as they are part of the istream class. In the example note especially the use of the member seekg:

```cpp
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
    istringstream istr("123 345"); // store some text.
    int x;

    istr.seekg(2); // skip "12"
    istr >> x; // extract int
    cout << x << '
'; // write it out
    istr.seekg(0); // retry from the beginning
    istr >> x; // extract int
    cout << x << '
'; // write it out
    istr.str("666"); // store another text
    istr >> x; // extract it
    cout << x << '
'; // write it out
}
/*
  output of this program:
  3
  123
  666
*/
```

### 6.5.4 Copying streams

Usually, files are copied either by reading a source file character by character or line by line. The basic mold to process streams is as follows:

- **Continuous loop:**
  1. read from the stream
  2. if reading did not succeed (i.e., fail returns true), break from the loop
  3. process the information that was read

Note that reading must precede testing, as it is only possible to know after actually attempting to read from a file whether the reading succeeded or not. Of course, variations are possible: getline(istream &, string &) (see section 6.5.1.1) returns an istream &, so here reading and testing may be contracted using one expression. Nevertheless, the above mold represents the general case. So, the following program may be used to copy cin to cout:

```cpp
#include <iostream>
using namespace std;

int main()
{
    while (true)
    {
```
6.5. INPUT

```cpp
{
  char c;
  cin.get(c);
  if (cin.fail())
    break;
  cout << c;
}
```

Contraction is possible here by combining `get` with the `if`-statement, resulting in:

```cpp
if (!cin.get(c))
  break;
```

Even so, this would still follow the basic rule: ‘read first, test later’.

Simply copying a file isn’t required very often. More often a situation is encountered where a file is processed up to a certain point, followed by plain copying the file’s remaining information. The next program illustrates this. Using `ignore` to skip the first line (for the sake of the example it is assumed that the first line is at most 80 characters long), the second statement uses yet another overloaded version of the `<<`-operator, in which a `streambuf` pointer is inserted into a stream. As the member `rdbuf` returns a stream’s `streambuf *`, we have a simple means of inserting a stream’s contents into an `ostream`:

```cpp
#include <iostream>
using namespace std;

int main()
{
  cin.ignore(80, '\n'); // skip the first line and...
  cout << cin.rdbuf(); // copy the rest through the streambuf *
}
```

This way of copying streams only assumes the existence of a `streambuf` object. Consequently it can be used with all specializations of the `streambuf` class.

### 6.5.5 Coupling streams

Ostream objects can be coupled to ios objects using the `tie` member function. Tying results in flushing the ostream’s buffer whenever an input or output operation is performed on the ios object to which the ostream object is tied. By default `cout` is tied to `cin` (using `cin.tie(cout)`). This tie means that whenever an operation on `cin` is requested, `cout` is flushed first. To break the tie, `ios::tie(0)` can be called. In the example: `cin.tie(0)`.

Another useful coupling of streams is shown by the tie between `cerr` and `cout`. Because of the tie standard output and error messages written to the screen are shown in sync with the time at which they were generated:

```cpp
#include <iostream>
using namespace std;
```
int main()
{
    cerr.tie(0);  // untie
    cout << "first (buffered) line to cout ";
    cerr << "first (unbuffered) line to cerr\n";
    cout << "\n";
    cerr.tie(&cout);  // tie cout to cerr
    cout << "second (buffered) line to cout ";
    cerr << "second (unbuffered) line to cerr\n";
    cout << "\n";
}
/*
   Generated output:
first (unbuffered) line to cerr
first (buffered) line to cout
second (buffered) line to cout second (unbuffered) line to cerr
*/

An alternative way to couple streams is to make streams use a common streambuf object. This can be implemented using the ios::rdbuf(streambuf *) member function. This way two streams can use, e.g. their own formatting, one stream can be used for input, the other for output, and redirection using the stream library rather than operating system calls can be implemented. See the next sections for examples.

### 6.6 Advanced topics

#### 6.6.1 Redirecting streams

Using ios::rdbuf streams can be forced to share their streambuf objects. Thus information written to one stream is actually written to another stream; a phenomenon normally called redirection. Redirection is commonly implemented at the operating system level, and sometimes that is still necessary (see section 24.2.3).

A common situation where redirection is useful is when error messages should be written to file rather than to the standard error stream, usually indicated by its file descriptor number 2. In the Unix operating system using the bash shell, this can be realized as follows:

```
program 2>&1
```

Following this command any error messages written by program are saved on the file /tmp/error.log, instead of appearing on the screen.

Here is an example showing how this can be implemented using streambuf objects. Assume program expects an argument defining the name of the file to write the error messages to. It could be called as follows:

```
program /tmp/error.log
```

The program looks like this, an explanation is provided below the program’s source text:
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char **argv)
{
    ofstream errlog; // 1
    streambuf *cerr_buffer = 0; // 2

    if (argc == 2)
    {
        errlog.open(argv[1]); // 3
        cerr_buffer = cerr.rdbuf(errlog.rdbuf()); // 4
    }
    else
    {
        cerr << "Missing log filename\n";
        return 1;
    }

cerr << "Several messages to stderr, msg 1\n";
cerr << "Several messages to stderr, msg 2\n";

cout << "Now inspect the contents of " << argv[1] << "... [Enter] ";
cin.get(); // 5

cerr << "Several messages to stderr, msg 3\n";

cerr.rdbuf(cerr_buffer); // 6

cerr << "Done\n"; // 7
}

/*
 Generated output on file argv[1]
 at cin.get():

 Several messages to stderr, msg 1
 Several messages to stderr, msg 2

 at the end of the program:

 Several messages to stderr, msg 1
 Several messages to stderr, msg 2
 Several messages to stderr, msg 3
 */

• At lines 1-2 local variables are defined: errlog is the ofstream to write the error messages too, and cerr_buffer is a pointer to a streambuf, to point to the original cerr buffer.

• At line 3 the alternate error stream is opened.

• At line 4 redirection takes place: cerr now writes to the streambuf defined by errlog. It is important that the original buffer used by cerr is saved, as explained below.
• At line 5 we pause. At this point, two lines were written to the alternate error file. We get a chance to take a look at its contents: there were indeed two lines written to the file.

• At line 6 the redirection is terminated. This is very important, as the `errlog` object is destroyed at the end of `main`. If `cerr`'s buffer would not have been restored, then at that point `cerr` would refer to a non-existing `streambuf` object, which might produce unexpected results. It is the responsibility of the programmer to make sure that an original `streambuf` is saved before redirection, and is restored when the redirection ends.

• Finally, at line 7, `Done` is again written to the screen, as the redirection has been terminated.

### 6.6.2 Reading AND Writing streams

In order to both read and write to a stream an `std::fstream` object must be created. As with `ifstream` and `ofstream` objects, its constructor receives the name of the file to be opened:

```cpp
fstream inout("iofile", ios::in | ios::out);
```

Note the use of the constants `ios::in` and `ios::out`, indicating that the file must be opened for both reading and writing. Multiple mode indicators may be used, concatenated by the `bitor` operator. Alternatively, instead of `ios::out`, `ios::app` could have been used and mere writing would become appending (at the end of the file).

Reading and writing to the same file is always a bit awkward: what to do when the file may not yet exist, but if it already exists it should not be rewritten? Having fought with this problem for some time I now use the following approach:

```cpp
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    fstream rw("fname", ios::out | ios::in);

    if (!rw) // file didn’t exist yet
    {
        rw.clear(); // try again, creating it using ios::trunc
        rw.open("fname", ios::out | ios::trunc | ios::in);
    }

    if (!rw) // can’t even create it: bail out
    {
        cerr << "Opening ‘fname’ failed miserably" << ‘\n’;
        return 1;
    }

    cerr << "We’re at: " << rw.tellp() << ‘\n’;
    // write something
    rw << "Hello world" << ‘\n’;
```
rw.seekg(0);       // go back and read what's written

string s;
getline(rw, s);

    cout << "Read: " << s << '\n';
}

Under this approach if the first construction attempt fails fname doesn't exist yet. But then open can be attempted using the ios::trunc flag. If the file already existed, the construction would have succeeded. By specifying ios::ate when defining rw, the initial read/write action would by default have taken place at EOF.

Under DOS-like operating systems that use the multiple character sequence \r\n to separate lines in text files the flag ios::binary is required to process binary files ensuring that \r\n combinations are processed as two characters. In general, ios::binary should be specified when binary (non-text) files are to be processed. By default files are opened as text files. Unix operating systems do not distinguish text files from binary files.

With fstream objects, combinations of file flags are used to make sure that a stream is or is not (re)created empty when opened. See section 6.4.2.1 for details.

Once a file has been opened in read and write mode, the << operator can be used to insert information into the file, while the >> operator may be used to extract information from the file. These operations may be performed in any order, but a seekg or seekp operation is required when switching between insertions and extractions. The seek operation is used to activate the stream's data used for reading or those used for writing (and vice versa). The istream and ostream parts of fstream objects share the stream's data buffer and by performing the seek operation the stream either activates its istream or its ostream part. If the seek is omitted, reading after writing and writing after reading simply fails. The example shows a white space delimited word being read from a file, writing another string to the file, just beyond the point where the just read word terminated. Finally yet another string is read which is found just beyond the location where the just written strings ended:

    fstream f("filename", ios::in | ios::out);
    string str;

    f >> str;       // read the first word

    // write a well known text
    f.seekg(0, ios::cur);
    f << "hello world";

    f.seekp(0, ios::cur);
    f >> str;       // and read again

Since a seek or clear operation is required when alternating between read and write (extraction and insertion) operations on the same file it is not possible to execute a series of << and >> operations in one expression statement.

Of course, random insertions and extractions are hardly ever used. Generally, insertions and extractions occur at well-known locations in a file. In those cases, the position where insertions or extractions are required can be controlled and monitored by the seekg, seekp, tellg and tellp members (see sections 6.4.1.2 and 6.5.1.2).
CHAPTER 6. THE IO-STREAM LIBRARY

Error conditions (see section 6.3.1) occurring due to, e.g., reading beyond end of file, reaching end of file, or positioning before begin of file, can be cleared by the clear member function. Following clear processing may continue. E.g.,

```cpp
fstream f("filename", ios::in | ios::out);
string str;

f.seekg(-10); // this fails, but...
f.clear(); // processing f continues
f >> str; // read the first word
```

A situation where files are both read and written is seen in database applications, using files consisting of records having fixed sizes, and where locations and sizes of pieces of information are known. For example, the following program adds text lines to a (possibly existing) file. It can also be used to retrieve a particular line, given its order-number in the file. A binary file index allows for the quick retrieval of the location of lines.

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <climits>
using namespace std;

void err(char const *msg)
{
    cout << msg << '
';
}

void err(char const *msg, long value)
{
    cout << msg << value << '
';
}

void read(fstream &index, fstream &strings)
{
    int idx;

    if (!cin >> idx) // read index
    {
        cin.clear(); // allow reading again
        cin.ignore(INT_MAX, '
'); // skip the line
        return err("line number expected");
    }

    index.seekg(idx * sizeof(long)); // go to index-offset

    long offset;

    if
    {
        !index.read // read the line-offset
        {
            reinterpret_cast<char *>(offset),
```
sizeof(long)
}
)
return err("no offset for line", idx);

if (!strings.seekg(offset)) // go to the line's offset
    return err("can't get string offset ", offset);

string line;
if (!getline(strings, line)) // read the line
    return err("no line at ", offset);

cout << "Got line: " << line << 'n'; // show the line
}

void write(fstream &index, fstream &strings)
{
    string line;
    if (!getline(cin, line)) // read the line
        return err("line missing");

    strings.seekp(0, ios::end); // to strings
    index.seekp(0, ios::end); // to index

    long offset = strings.tellp();

    if
    {
        !index.write // write the offset to index
        {
            reinterpret_cast<char*>(offset),
            sizeof(long)
        }
    }
    return err("Writing failed to index: ", offset);

    if (!strings << line << 'n') // write the line itself
        return err("Writing to 'strings' failed");
    // confirm writing the line
    cout << "Write at offset " << offset << " line: " << line << 'n';
}

int main()
{
    fstream index("index", ios::trunc | ios::in | ios::out);
    fstream strings("strings", ios::trunc | ios::in | ios::out);

    cout << "enter 'r <number>' to read line <number> or "
    "w <line>' to write a line\n"
    "or enter 'q' to quit.\n";

    while (true)
Another example showing reading and writing of files is provided by the next program. It also illustrates the processing of NTB strings:

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    // r/w the file
    fstream f("hello", ios::in | ios::out | ios::trunc);
    f.write("hello", 6); // write 2 NTB strings
    f.write("hello", 6);
    f.seekg(0, ios::beg); // reset to begin of file
    char buffer[100]; // or: char *buffer = new char[100]
    char c;
    // read the first 'hello'
    cout << f.get(buffer, sizeof(buffer), 0).tellg() << '\n';
    f >> c; // read the NTB delim
    // and read the second 'hello'
    cout << f.get(buffer + 6, sizeof(buffer) - 6, 0).tellg() << '\n';
    buffer[5] = ' '; // change asciiz to ' ' 
    cout << buffer << '\n'; // show 2 times 'hello'
}``
A completely different way to read and write streams may be implemented using `streambuf` members. All considerations mentioned so far remain valid (e.g., before a read operation following a write operation `seekg` must be used). When `streambuf` objects are used, either an `istream` is associated with the `streambuf` object of another `ostream` object, or an `ostream` object is associated with the `streambuf` object of another `istream` object. Here is the previous program again, now using associated streams:

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

void err(char const *msg); // see earlier example
void err(char const *msg, long value);

void read(istream &index, istream &strings)
{
    index.clear();
    strings.clear();

    // insert the body of the read() function of the earlier example
}

void write(ostream &index, ostream &strings)
{
    index.clear();
    strings.clear();

    // insert the body of the write() function of the earlier example
}

int main()
{
    ifstream index_in("index", ios::trunc | ios::in | ios::out);
    ifstream strings_in("strings", ios::trunc | ios::in | ios::out);
    ofstream index_out(index_in.rdbuf());
    ofstream strings_out(strings_in.rdbuf());

    cout << "enter 'r <number>' to read line <number> or "
        "w <line>' to write a line\n"
        "or enter 'q' to quit.\n";

    while (true)
    {
        cout << "r <nr>, w <line>, q ? "; // show prompt
```
string cmd;

    cin >> cmd;           // read cmd
    if (cmd == "q")       // process the cmd.
        return 0;
    if (cmd == "r")
        read(index_in, strings_in);
    else if (cmd == "w")
        write(index_out, strings_out);
    else
        cout << "Unknown command: " << cmd << '
';
}

In this example

- the streams associated with the streambuf objects of existing streams are not ifstream or ofstream objects but basic istream and ostream objects.

- The streambuf object is not defined by an ifstream or ofstream object. Instead it is defined outside of the streams, using a filebuf (cf. section 14.8.2) and constructions like:

  ```cpp
  filebuf fb("index", ios::in | ios::out | ios::trunc);
  istream index_in(&fb);
  ostream index_out(&fb);
  ```

- An ifstream object can be constructed using stream modes normally used with ofstream objects. Conversely, an ofstream objects can be constructed using stream modes normally used with ifstream objects.

- If istream and ostream objects share a streambuf, then their read and write pointers (should) point to the shared buffer: they are tightly coupled.

- The advantage of using an external (separate) streambuf over a predefined fstream object is (of course) that it opens the possibility of using stream objects with specialized streambuf objects. These streambuf objects may specifically be constructed to control and interface particular devices. Elaborating this (see also section 14.8) is left as an exercise to the reader.
Chapter 7

Classes

The C programming language offers two methods for structuring data of different types. The C struct holds data members of various types, and the C union also defines data members of various types. However, a union’s data members all occupy the same location in memory and the programmer may decide on which one to use.

In this chapter classes are introduced. A class is a kind of struct, but its contents are by default inaccessible to the outside world, whereas the contents of a C++ struct are by default accessible to the outside world. In C++ structs find little use: they are mainly used to aggregate data within the context of classes or to define elaborate return values. Often a C++ struct merely contains plain old data (POD, cf. section 9.9). In C++ the class is the main data structuring device, by default enforcing two core concepts of current-day software engineering: data hiding and encapsulation (cf. sections 3.2.1 and 7.1.1).

The union is another data structuring device the language offers. The traditional C union is still available in C++, but the C++11 standard adds unrestricted unions to the language. Unrestricted unions are unions whose data fields may be of class types. The C++ Annotations covers these unrestricted unions in section 12.6, after having introduced several other new concepts of C++.

C++ extends the C struct and union concepts by allowing the definition of member functions (introduced in this chapter) within these data types. Member functions are functions that can only be used with objects of these data types or within the scope of these data types. Some of these member functions are special in that they are always, usually automatically, called when an object starts its life (the so-called constructor) or ends its life (the so-called destructor). These and other types of member functions, as well as the design and construction of, and philosophy behind, classes are introduced in this chapter.

We step-by-step construct a class Person, which could be used in a database application to store a person’s name, address and phone number.

Let’s start by creating a class Person right away. From the onset, it is important to make the distinction between the class interface and its implementation. A class may loosely be defined as ‘a set of data and all the functions operating on those data’. This definition is later refined but for now it is sufficient to get us started.

A class interface is a definition, defining the organization of objects of that class. Normally a definition results in memory reservation. E.g., when defining int variable the compiler ensures that some memory is reserved in the final program storing variable’s values. Although it is a definition no memory is set aside by the compiler once it has processed the class definition. But a class definition follows the one definition rule: in C++ entities may be defined only once. As a class definition
does not imply that memory is being reserved the term class interface is preferred instead.

Class interfaces are normally contained in a class header file, e.g., person.h. We’ll start our class Person interface here (cf section 7.7 for an explanation of the const keywords behind some of the class’s member functions):

```cpp
#include <string>

class Person
{
  std::string d_name; // name of person
  std::string d_address; // address field
  std::string d_phone;  // telephone number
  size_t d_mass;       // the mass in kg.

public: // member functions
  void setName(std::string const &name);
  void setAddress(std::string const &address);
  void setPhone(std::string const &phone);
  void setMass(size_t mass);
  std::string const &name() const;
  std::string const &address() const;
  std::string const &phone() const;
  size_t mass() const;
};
```

The member functions that are declared in the interface must still be implemented. The implementation of these members is properly called their definition.

In addition to the member function a class defines the data manipulated by the member functions. These data are called the data member data members. In Person they are d_name, d_address, d_phone and d_mass. Data members should be given private access rights. Since the class uses private access rights by default they may simply be listed at the top of the interface.

All communication between the outer world and the class data is routed through the class’s member functions. Data members may receive new values (e.g., using setName) or they may be retrieved for inspection (e.g., using name). Functions merely returning values stored inside the object, not allowing the caller to modify these internally stored values, are called accessors.

Syntactically there is only a marginal difference between a class and a struct. Classes by default define private members, structs define public members. Conceptually, though, there are differences. In C++ structs are used in the way they are used in C: to aggregate data, which are all freely accessible. Classes, on the other hand, hide their data from access by the outside world (which is aptly called data hiding) and offer member functions to define the communication between the outer world and the class’s data members.

Following Lakos (Lakos, J., 2001) Large-Scale C++ Software Design (Addison-Wesley) I suggest the following setup of class interfaces:

- All data members have private access rights, and are placed at the top of the interface.
- All data members start with d_, followed by a name suggesting their meaning (in chapter 8 we’ll also encounter data members starting with s_).
- Non-private data members do exist, but one should be hesitant to define non-private access rights for data members (see also chapter 13).
Two broad categories of member functions are manipulators and accessors. Manipulators allow the users of objects to modify the internal data of the objects. By convention, manipulators start with set. E.g., setName.

With accessors, a get-prefix is still frequently encountered, e.g., getName. However, following the conventions promoted by the Qt (see http://www.trolltech.com) Graphical User Interface Toolkit, the get-prefix is now deprecated. So, rather than defining the member getAddress, it should simply be named address.

Normally (exceptions exist) the public member functions of a class are listed first, immediately following the class’s data members. They are the important elements of the interface as they define the features the class is offering to its users. It’s a matter of convention to list them high up in the interface. The keyword private is needed beyond the public members to switch back from public members to private access rights which nicely separates the members that may be used ‘by the general public’ from the class’s own support members.

Style conventions usually take a long time to develop. There is nothing obligatory about them, though. I suggest that readers who have compelling reasons not to follow the above style conventions use their own. All others are strongly advised to adopt the above style conventions.

Finally, referring back to section 3.1.1 that

```cpp
using namespace std;
```

must be used in most (if not all) examples of source code. As explained in sections 7.12 and 7.12.1 the using directive should follow the preprocessor directive(s) including the header files, using a setup like the following:

```cpp
#include <iostream>
#include "person.h"

using namespace std;

int main()
{
    ...
}
```

### 7.1 The constructor

C++ classes may contain two special categories of member functions which are essential to the proper working of the class. These categories are the constructors and the destructor. The destructor’s primary task is to return memory allocated by an object to the common pool when an object goes ‘out of scope’. Allocation of memory is discussed in chapter 9, and destructors are therefore be discussed in depth in that chapter. In this chapter the emphasis is on the class’s organization and its constructors.

Constructor are recognized by their names which is equal to the class name. Constructors do not specify return values, not even void. E.g., the class Person may define a constructor `Person::Person()`. The C++ run-time system ensures that the constructor of a class is called when a variable of the class is defined. It is possible to define a class lacking any constructor. In that case the compiler defines a default constructor that is called when an object of that class is defined.
What actually happens in that case depends on the data members that are defined by that class (cf. section 7.3.1).

Objects may be defined locally or globally. However, in C++ most objects are defined locally. Globally defined objects are hardly ever required and are somewhat deprecated.

When a local object is defined its constructor is called every time the function is called. The object’s constructor is activated at the point where the object is defined (a subtlety is that an object may be defined implicitly as, e.g., a temporary variable in an expression).

When an object is defined as a static object, it is constructed when the program starts. In this case its constructor is called even before the function main starts. Example:

```cpp
#include <iostream>
using namespace std;

class Demo
{
   public:
      Demo();
   }

Demo::Demo()
{
   cout << "Demo constructor called\n";
}

Demo d;

int main()
{
}

/*
   Generated output:
   Demo constructor called
*/
```

The program contains one global object of the class Demo with main having an empty body. Nonetheless, the program produces some output generated by the constructor of the globally defined Demo object.

Constructors have a very important and well-defined role. They must ensure that all the class’s data members have sensible or at least well-defined values once the object has been constructed. We’ll get back to this important task shortly. The default constructor has no argument. It is defined by the compiler unless another constructor is defined and unless its definition is suppressed (cf. section 7.6). If a default constructor is required in addition to another constructor then the default constructor must explicitly be defined as well. The C++11 standard provides special syntax to do that as well, which is also covered by section 7.6.

7.1.1 A first application

Our example class Person has three string data members and a size_t d_mass data member. Access to these data members is controlled by interface functions.
Whenever an object is defined the class’s constructor(s) ensure that its data members are given sensible values. Thus, objects never suffer from uninitialized values. Data members may be given new values, but that should never be directly allowed. It is a core principle (called data hiding) of good class design that its data members are private. The modification of data members is therefore fully controlled by member functions and thus, indirectly, by the class-designer. The class encapsulates all actions performed on its data members and due to this encapsulation the class object may assume the responsibility for its own data-integrity. Here is a minimal definition of Person’s manipulating members:

```cpp
#include "person.h" // given earlier
using namespace std;

void Person::setName(string const &name) {
    d_name = name;
}
void Person::setAddress(string const &address) {
    d_address = address;
}
void Person::setPhone(string const &phone) {
    d_phone = phone;
}
void Person::setMass(size_t mass) {
    d_mass = mass;
}
```

It’s a minimal definition in that no checks are performed. But it should be clear that checks are easy to implement. E.g., to ensure that a phone number only contains digits one could define:

```cpp
void Person::setPhone(string const &phone) {
    if (phone.find_first_not_of("0123456789") == string::npos)
        d_phone = phone;
    else
        cout << "A phone number may only contain digits\n";
}
```

Similarly, access to the data members is controlled by encapsulating accessor members. Accessors ensure that data members cannot suffer from uncontrolled modifications. Since accessors conceptually do not modify the object’s data (but only retrieve the data) these member functions are given the predicate const. They are called const member functions, which, as they are guaranteed not to modify their object’s data, are available to both modifiable and constant objects (cf. section 7.7).

To prevent backdoors we must also make sure that the data member is not modifiable through an accessor’s return value. For values of built-in primitive types that’s easy, as they are usually returned by value, which are copies of the values found in variables. But since objects may be fairly large making copies are usually prevented by returning objects by reference. A backdoor is created by returning a data member by reference, as in the following example, showing the allowed abuse below the function definition:

```cpp
string &Person::name() const
```
```cpp
{  
  return d_name;
}

Person somebody;
somebody.setName("Nemo");

somebody.name() = "Eve";  // Oops, backdoor changing the name

To prevent the backdoor objects are returned as const references from accessors. Here are the implementations of Person's accessors:

```cpp
#include "person.h"  // given earlier
using namespace std;

string const &Person::name() const
{  
  return d_name;
}

string const &Person::address() const
{  
  return d_address;
}

string const &Person::phone() const
{  
  return d_phone;
}

size_t Person::mass() const
{  
  return d_mass;
}
```

The Person class interface remains the starting point for the class design: its member functions define what can be asked of a Person object. In the end the implementation of its members merely is a technicality allowing Person objects to do their jobs.

The next example shows how the class Person may be used. An object is initialized and passed to a function printperson(), printing the person's data. Note the reference operator in the parameter list of the function printperson. Only a reference to an existing Person object is passed to the function, rather than a complete object. The fact that printperson does not modify its argument is evident from the fact that the parameter is declared const.

```cpp
#include <iostream>
#include "person.h"  // given earlier
using namespace std;

void printperson(Person const &p)
{
  cout << "Name : " << p.name() << "\n"  
      "Address : " << p.address() << "\n"  
      "Phone : " << p.phone() << "\n"  
      "Mass : " << p.mass() << "\n";
}
7.1. THE CONSTRUCTOR

```cpp
int main()
{
    Person p;
    p.setName("Linus Torvalds");
p.setAddress("E-mail: Torvalds@cs.helsinki.fi");
p.setPhone(" - not sure -");
p.setMass(75); // kg.
p.setMass(75);
    printperson(p);
}
/*
Produced output:
Name : Linus Torvalds
Address : E-mail: Torvalds@cs.helsinki.fi
Phone : - not sure -
Mass : 75
*/
```

7.1.2 Constructors: with and without arguments

The class `Person`'s constructor so far has no parameters. C++ allows constructors to be defined with or without parameter lists. The arguments are supplied when an object is defined.

For the class `Person` a constructor expecting three strings and a `size_t` might be useful. Representing, respectively, the person's name, address, phone number and mass. This constructor is (but see also section 7.3.1):

```cpp
Person::Person(string const &name, string const &address,
               string const &phone, size_t mass)
{
    d_name = name;
    d_address = address;
    d_phone = phone;
    d_mass = mass;
}
```

It must of course also be declared in the class interface:

```cpp
class Person
{
    // data members (not altered)

    public:
        Person(std::string const &name, std::string const &address,
               std::string const &phone, size_t mass);

        // rest of the class interface (not altered)
};
```

Now that this constructor has been declared, the default constructor must explicitly be declared as
well if we still want to be able to construct a plain Person object without any specific initial values for its data members. The class Person would thus support two constructors, and the part declaring the constructors now becomes:

```cpp
class Person
{
    // data members
    public:
        Person();
        Person(std::string const &name, std::string const &address,
               std::string const &phone, size_t mass);

    // additional members
};
```

In this case, the default constructor doesn’t have to do very much, as it doesn’t have to initialize the string data members of the Person object. As these data members are objects themselves, they are initialized to empty strings by their own default constructor. However, there is also a size_t data member. That member is a variable of a built-in type and such variables do not have constructors and so are not initialized automatically. Therefore, unless the value of the d_mass data member is explicitly initialized its value is:

- a random value for local Person objects;
- 0 for global and static Person objects.

The 0-value might not be too bad, but normally we don’t want a random value for our data members. So, even the default constructor has a job to do: initializing the data members which are not initialized to sensible values automatically. Its implementation can be:

```cpp
Person::Person()
{
    d_mass = 0;
}
```

Using constructors with and without arguments is illustrated next. The object karel is initialized by the constructor defining a non-empty parameter list while the default constructor is used with the anon object:

```cpp
int main()
{
    Person karel("Karel", "Rietveldlaan 37", "542 6044", 70);
    Person anon;
}
```

The two Person objects are defined when main starts as they are local objects, living only for as long as main is active.

If Person objects must be definable using other arguments, corresponding constructors must be added to Person’s interface. Apart from overloading class constructors it is also possible to provide constructors with default argument values. These default arguments must be specified with the constructor declarations in the class interface, like so:

```cpp
class Person
```
7.1. THE CONSTRUCTOR

```cpp
{
    public:
        Person(std::string const &name,
            std::string const &address = "--unknown--",
            std::string const &phone = "--unknown--",
            size_t mass = 0);
};
```

Often, constructors use highly similar implementations. This results from the fact that the constructor's parameters are often defined for convenience: a constructor not requiring a phone number but requiring a mass cannot be defined using default arguments, since phone is not the constructor's last parameter. Consequently a special constructor is required not having phone in its parameter list.

In pre C++11 C++ this situation is commonly tackled as follows: all constructors must initialize their reference and const data members, or the compiler (rightfully) complains. To initialize the remaining members (non-const and non-reference members) we have two options:

- If the body of the construction process is sizeable but (parameterizable) identical to other constructors bodies then factorize. Define a private member init which is called by the constructors to provide the object's data members with their appropriate values.
- If the constructors act fundamentally differently, then there's nothing to factorize and each constructor must be implemented by itself.

C++11 allows constructors to call each other. This is illustrated in section 7.4.1 below.

7.1.2.1 The order of construction

The possibility to pass arguments to constructors allows us to monitor the construction order of objects during program execution. This is illustrated by the next program using a class Test. The program defines a global Test object and two local Test objects. The order of construction is as expected: first global, then main's first local object, then func's local object, and then, finally, main's second local object:

```cpp
#include <iostream>
#include <string>
using namespace std;

class Test
{
    public:
        Test(string const &name); // constructor with an argument
    
};

Test::Test(string const &name)
{
    cout << "Test object " << name << " created" << '\n';
}

Test globaltest("global");
```
7.2 Ambiguity resolution

Defining objects may result in some unexpected surprises. Assume the following class interface is available:

```cpp
class Data {
    public:
        Data();
        Data(int one);
        Data(int one, int two);
        void display();
};
```

The intention is to define two objects of the class Data, using, respectively, the first and second constructors. Your code looks like this (and compiles correctly):

```cpp
#include "data.h"
int main()
{
    Data d1();
    Data d2(argc);
}
```

Now it's time to make some good use of the `Data` objects. You add two statements to `main`:

```cpp
d1.display();
d2.display();
```

But, surprise, the compiler complains about the first of these two:
error: request for member 'display' in 'd1', which is of non-class type 'Data()'

What's going on here? First of all, notice the data type the compiler refers to: \texttt{Data()}, rather than \texttt{Data}. What are those \texttt{()} doing there?

Before answering that question, let's broaden our story somewhat. We know that somewhere in a library a factory function \texttt{dataFactory} exists. A factory function creates and returns an object of a certain type. This \texttt{dataFactory} function returns a \texttt{Data} object, constructed using \texttt{Data}'s default constructor. Hence, \texttt{dataFactory} needs no arguments. We want to use \texttt{dataFactory} in our program, but must declare the function. So we add the declaration to \texttt{main}, as that's the only location where \texttt{dataFactory} will be used. It's a function, not requiring arguments, returning a \texttt{Data} object:

\begin{verbatim}
Data dataFactory();
\end{verbatim}

This, however, looks remarkably similar to our \texttt{d1} object definition:

\begin{verbatim}
Data d1();
\end{verbatim}

We found the source of our problem: \texttt{Data d1()} apparently is \textit{not} the definition of a \texttt{d1} object, but the \textit{declaration} of a function, returning a \texttt{Data} object. So, what's happening here and how should we define a \texttt{Data} object using \texttt{Data}'s default constructor?

First: what's happening here is that the compiler, when confronted with \texttt{Data d1()}, actually had a choice. It could either define a \texttt{Data} object, or declare a function. It declares a function.

In fact, we're encountering an ambiguity in \texttt{C++}'s syntax here, which is solved, according to the language's standard, by always letting a declaration prevail over a definition. We'll encounter more situations where this ambiguity occurs later on in this section.

Second: there are several ways we can solve this ambiguity the way we want it to be solved. To define an object using its default constructor:

\begin{itemize}
  \item merely mention it (like \texttt{int x}): \texttt{Data d1;}
  \item use the curly-brace initialization: \texttt{Data d1{}};
  \item use the assignment operator and an anonymous default constructed \texttt{Data} object: \texttt{Data d1 = Data().}
\end{itemize}

7.2.1 \textbf{Types 'Data' vs. 'Data()'}

\texttt{Data()}, which in the above context defines a default constructed anonymous \texttt{Data} object, takes us back to the compiler error. According to the compiler, our original \texttt{d1} apparently was not of type \texttt{Data}, but of type \texttt{Data()}. So what's that?

Let's first have a look at our second constructor. It expects an \texttt{int}. We would like to define another \texttt{Data} object, using the second constructor, but want to pass the default \texttt{int} value to the constructor, using \texttt{int{}}. We know this defines a default \texttt{int} value, as \texttt{cout « int{}} « ' \texttt{\textbackslash n} ' \texttt{nicely displays 0, and int x = int{}} also initialized \texttt{x} to 0. So we define \texttt{`Data di(int{})'} in \texttt{main}.

Not good: again the compiler complains when we try to use \texttt{di}. After \texttt{`di.display()'} the compiler tells us:

\begin{verbatim}
error: request for member 'display' in 'di', which is of non-class type 'Data(int (*)())'
\end{verbatim}
Oops, not as expected.... Didn’t we pass 0? Why the sudden pointer? It’s that same ‘use a declaration when possible’ strategy again. The notation Type() not only represents the default value of type Type, but it’s also a shorthand notation for an anonymous pointer to a function, not expecting arguments, and returning a Type value, which you can verify by defining ‘int (*ip)() = nullptr’, and passing ip as argument to di: di(ip) compiles fine.

So why doesn’t the error occur when inserting int() or assigning int() to int x? In these latter cases nothing is declared. Rather, cout and int x = need expressions determining values, which is provided by int()’s ‘natural’ interpretation. But with ‘Data di(int())’ the compiler again has a choice, and (by design) it chooses a declaration because the declaration takes priority. Now int()’s interpretation as an anonymous pointer is available and therefore used.

Likewise, if int x has been defined, ‘Data b1(int(x))’ declares b1 as a function, expecting an int (as int(x) represents a type), while ‘Data b2((int)x)’ defines b2 as a Data object, using the constructor expecting a single int value.

### 7.2.2 Superfluous parentheses

Let’s play some more. At some point in our program we defined int b. Then, in a compound statement we need to construct an anonymous Data object, initialized using b, followed by displaying b:

```cpp
int b = 18;
{
  Data(b);
  cout << b;
}
```

About that cout statement the compiler tells us (I modified the error message to reveal its meaning):

*error: cannot bind ‘std::ostream & « Data const &’*

Here we didn’t insert int b but Data b. Had we omitted the compound statement, the compiler would have complained about a doubly defined b entity, as Data(b) simply means Data b, a Data object constructed by default. The compiler may omit superfluous parentheses when parsing a definition or declaration.

Of course, the question now becomes how a temporary object Data, initialized with int b can be defined. Remember that the compiler may remove superfluous parentheses. So, what we need to do is to pass an int to the anonymous Data object, without using the int’s name.

- We can use a cast: Data(static_cast<int>(b));
- We can use a curly-brace initialization: Data {b}.

Values and types make big differences. Consider the following definitions:

```cpp
Data (*d4)(int);  // 1
Data (*d5)(3);    // 2
```

Definition 1 should cause no problems: it’s a pointer to a function, expecting an int, returning a Data object. Hence, d4 is a pointer variable.
Definition 2 is slightly more complex. Yes, it’s a pointer. But it has nothing to do with a function. So what’s that argument list containing 3 doing there? Well, it’s not an argument list. It’s an initialization that looks like an argument list. Remember that variables can be initialized using the assignment statement, by parentheses or by curly parentheses. So instead of ‘(3)’ we could have written ‘= 3’ or ‘{3}’. Let’s pick the first alternative, resulting in:

```c
Data (*d5) = 3;
```

Now we get to ‘play compiler’ again. Removing some superfluous parentheses we get:

```c
Data *d5 = 3;
```

It’s a pointer to a Data object, initialized to 3 (semantically incorrect, but that’s only clear after the syntactical analysis. If I had initially written

```c
Data (*d5)(&d1); // 2
```

the fun resulting from contrasting int and 3 would most likely have been spoiled).

### 7.2.3 Existing types

Once a type name has been defined it also prevails over identifiers representing variables, if the compiler is given a choice. This, too, can result in interesting constructions.

Assume a function process expecting an int exists in a library. We want to use this function to process some int data values. So in main process is declared and called:

```c
int process(int Data);
process(argc);
```

No problems here. But unfortunately we once decided to ‘beautify’ our code, by throwing in some superfluous parentheses, like so:

```c
int process(int (Data));
process(argc);
```

Now we’re in trouble. The compiler now generates an error, caused by its rule to let declarations prevail over definitions. Data now becomes the name of the class Data, and analogous to int (x) the parameter int (Data) is parsed as int (*)(Data): a pointer to a function, expecting a Data object, returning an int.

Here is another example. When, instead of declaring

```c
int process(int Data[10]);
```

we declare, e.g., to emphasize the fact that an array is passed to process:

```c
int process(int (Data[10]));
```
the `process` function does not expect a pointer to `int` values, but a pointer to a function expecting a pointer to `Data` elements, returning an `int`.

To summarize the findings in the ‘Ambiguity Resolution’ section:

- The compiler will try to remove superfluous parentheses;
- But if the parenthesized construction represents a type, it will try to use the type;
- More in general: when possible the compiler will interpret a syntactic construction as a declaration, rather than as a definition (of an object or variable).

### 7.3 Objects inside objects: composition

In the class `Person` objects are used as data members. This construction technique is called **composition**.

Composition is neither extraordinary nor C++ specific: in C a `struct` or `union` field is commonly used in other compound types. In C++ it requires some special thought as their initialization sometimes is subject to restrictions, as discussed in the next few sections.

#### 7.3.1 Composition and `const` objects: `const` member initializers

Unless specified otherwise object data members of classes are initialized by their default constructors. Using the default constructor might not always be the optimal way to initialize an object and it might not even be possible: a class might simply not define a default constructor.

Earlier we’ve encountered the following constructor of the `Person`:

```cpp
Person::Person(string const &name, string const &address,
               string const &phone, size_t mass)
{
    d_name = name;
    d_address = address;
    d_phone = phone;
    d_mass = mass;
}
```

Think briefly about what is going on in this constructor. In the constructor’s body we encounter assignments to string objects. Since assignments are used in the constructor’s body their left-hand side objects must exist. But when objects are coming into existence constructors must have been called. The initialization of those objects is thereupon immediately undone by the body of `Person`’s constructor. That is not only inefficient but sometimes downright impossible. Assume that the class interface mentions a `string const` data member: a data member whose value is not supposed to change at all (like a birthday, which usually doesn’t change very much and is therefore a good candidate for a `string const` data member). Constructing a birthday object and providing it with an initial value is OK, but changing the initial value isn’t.

The body of a constructor allows assignments to data members. The initialization of data members happens before that. C++ defines the **member initializer** syntax allowing us to specify the way data members are initialized at construction time. Member initializers are specified as a list of
7.3. **OBJECTS INSIDE OBJECTS: COMPOSITION**

Constructor specifications between a colon following a constructor's parameter list and the opening curly brace of a constructor's body, as follows:

```
Person::Person(string const &name, string const &address,
               string const &phone, size_t mass)
  :
    d_name(name),
    d_address(address),
    d_phone(phone),
    d_mass(mass)
{
}
```

Member initialization *always* occurs when objects are composed in classes: if no constructors are mentioned in the member initializer list the default constructors of the objects are called. Note that this only holds true for objects. Data members of primitive data types are *not* initialized automatically.

Member initialization can, however, also be used for primitive data members, like `int` and `double`. The above example shows the initialization of the data member `d_mass` from the parameter `mass`. When member initializers are used the data member could even have the same name as the constructor's parameter (although this is deprecated) as there is no ambiguity and the first (left) identifier used in a member initializer is always a data member that is initialized whereas the identifier between parentheses is interpreted as the parameter.

The *order* in which class type data members are initialized is defined by the order in which those members are defined in the composing class interface. If the order of the initialization in the constructor differs from the order in the class interface, the compiler complains, and reorders the initialization so as to match the order of the class interface.

Member initializers should be used as often as possible. As shown it may be required to use them (e.g., to initialize const data members, or to initialize objects of classes lacking default constructors) but *not* using member initializers also results in inefficient code as the default constructor of a data member is always automatically called unless an explicit member initializer is specified. Reassignment in the constructor's body following default construction is then clearly inefficient. Of course, sometimes it is fine to use the default constructor, but in those cases the explicit member initializer can be omitted.

As a rule of thumb: if a value is assigned to a data member in the constructor's body then try to avoid that assignment in favor of using a member initializer.

### 7.3.2 Composition and reference objects: reference member initializers

Apart from using member initializers to initialize composed objects (be they `const` objects or not), there is another situation where member initializers must be used. Consider the following situation.

A program uses an object of the class `Configfile`, defined in `main` to access the information in a configuration file. The configuration file contains parameters of the program which may be set by changing the values in the configuration file, rather than by supplying command line arguments.

Assume another object used in `main` is an object of the class `Process`, doing ‘all the work’. What possibilities do we have to tell the object of the class `Process` that an object of the class `Configfile` exists?

- The objects could have been declared as *global* objects. This is a possibility, but not a very good
one, since all the advantages of local objects are lost.

- The `Configfile` object may be passed to the `Process` object at construction time. Bluntly passing an object (i.e., by value) might not be a very good idea, since the object must be copied into the `Configfile` parameter, and then a data member of the `Process` class can be used to make the `Configfile` object accessible throughout the `Process` class. This might involve yet another object-copying task, as in the following situation:

```cpp
Process::Process(Configfile conf) // a copy from the caller
{
    d_conf = conf; // copying to d_conf member
}
```

- The copy-instructions can be avoided if `pointers` to the `Configfile` objects are used, as in:

```cpp
Process::Process(Configfile *conf) // pointer to external object
{
    d_conf = conf; // d_conf is a Configfile *
}
```

This construction as such is OK, but forces us to use the ‘->’ field selector operator, rather than the ‘.’ operator, which is (disputably) awkward. Conceptually one tends to think of the `Configfile` object as an object, and not as a pointer to an object. In C this would probably have been the preferred method, but in C++ we can do better.

- Rather than using value or pointer parameters, the `Configfile` parameter could be defined as a `reference parameter of Process's constructor`. Next, use a `Config` reference data member in the class `Process`.

But a reference variable cannot be initialized using an assignment, and so the following is incorrect:

```cpp
Process::Process(Configfile &conf)
{
    d_conf = conf; // wrong: no assignment
}
```

The statement `d_conf = conf` fails, because it is not an initialization, but an assignment of one `Configfile` object (i.e., `conf`), to another (`d_conf`). An assignment to a reference variable is actually an assignment to the variable the reference variable refers to. But which variable does `d_conf` refer to? To no variable at all, since we haven’t initialized `d_conf`. After all, the whole purpose of the statement `d_conf = conf` was to initialize `d_conf`....

How to initialize `d_conf`? We once again use the member initializer syntax. Here is the correct way to initialize `d_conf`:

```cpp
Process::Process(Configfile &conf)
    : d_conf(conf) // initializing reference member
{
}
```

The above syntax must be used in all cases where reference data members are used. E.g., if `d_ir` would have been an `int` reference data member, a construction like

```cpp
Process::Process(int &ir)
```
7.4 Data member initializers

Non-static data members of classes are usually initialized by the class's constructors. Frequently (but not always) the same initializations are used by different constructors, resulting in multiple points where the initializations are performed, which in turn complicates class maintenance.

Consider a class defining several data members: a pointer to data, a data member storing the number of data elements the pointer points at, a data member storing the sequence number of the object. The class also offers a basic set of constructors, as shown in the following class interface:

```cpp
class Container
{
    Data *d_data;
    size_t d_size;
    size_t d_nr;

    static size_t s_nObjects;

public:
    Container();
    Container(Container const &other);
    Container(Data *data, size_t size);
    Container(Container &&tmp);
};
```

The initial values of the data members are easy to describe, but somewhat hard to implement. Consider the initial situation and assume the default constructor is used: all data members should be set to 0, except for \(d_{nr}\) which must be given the value \(++s_{nObjects}\). Since these are non-default actions, we can't declare the default constructor using `= default`, but we must provide an actual implementation:

```cpp
Container()
:
    d_data(0),
    d_size(0),
    d_nr(++s_nObjects)
{};
```

In fact, all constructors require us to state the \(d_{nr}(++s_{nObjects})\) initialization. So if \(d_{data}\)'s type would have been a (move aware) class type, we would still have to provide implementations for all of the above constructors.

The C++11 standard, however, supports data member initializers, simplifying the initialization of non-static data members. Data member initializers allow us to assign initial values to data members. The compiler must be able to compute these initial values from initialization expressions, but the initial values do not have to be constant expressions. So \(++s_{nObjects}\) can be an initial value.
Using data member initializers for the class `Container` we get:

```cpp
class Container
{
    Data *d_data = 0;
    size_t d_size = 0;
    size_t d_nr = ++nObjects;

    static size_t s_nObjects;

public:
    Container() = default;
    Container(Container const &other);
    Container(Data *data, size_t size);
    Container(Container &&tmp);
};
```

Note that the data member initializations are recognized by the compiler, and are applied to its implementation of the default constructor. In fact, all constructors will apply the data member initializations, unless explicitly initialized otherwise. E.g., the move-constructor may now be implemented like this:

```cpp
Container(Container &&tmp)
{
    d_data(tmp.d_data),
    d_size(tmp.d_size)
    { tmp.d_data = 0;
}
```

Although `d_nr`’s initialization is left out of the implementation it is initialized due to the data member initialization provided in the class’s interface.

### 7.4.1 Delegating constructors

Often constructors are specializations of each other, allowing objects to be constructed specifying only subsets of arguments for all of its data members, using default argument values for the remaining data members.

Before the C++11 standard common practice was to define a member like `init` performing all initializations common to constructors. Such an `init` function, however, cannot be used to initialize `const` or reference data members, nor can it be used to perform so-called base class initializations (cf. chapter 13).

Here is an example where such an `init` function might have been used. A class `Stat` is designed as a wrapper class around C’s `stat(2)` function. The class might define three constructors: one expecting no arguments and initializing all data members to appropriate values; a second one doing the same, but it calls `stat` for the filename provided to the constructor; and a third one expecting a filename and a search path for the provided file name. Instead of repeating the initialization code in each constructor, the common code can be factorized into a member `init` which is called by the constructors.

The C++11 standard offers an alternative by allowing constructors to call each other. This is called
7.5 Uniform initialization

delegating constructors The C++11 standard allows us to delegate constructors as illustrated by the next example:

```cpp
class Stat
{
    public:
        Stat()
            :
                State("", ")  // no filename/searchpath
        {}
        Stat(std::string const &fileName)
            :
                Stat(fileName, ")  // only a filename
        {}
        Stat(std::string const &fileName, std::string const &searchPath)
            :
                d_filename(fileName),
                d_searchPath(searchPath)
        {
            // remaining actions to be performed by the constructor
        }
};
```

C++ allows static const integral data members to be initialized within the class interfaces (cf. chapter 8). The C++11 standard adds to this the facility to define default initializations for plain data members in class interfaces (these data members may or may not be const or of integral types, but (of course) they cannot be reference data members).

These default initializations may be overruled by constructors. E.g., if the class Stat uses a data member bool d_hasPath which is false by default but the third constructor (see above) should initialize it to true then the following approach is possible:

```cpp
class Stat
{
    bool d_hasPath = false;

    public:
        Stat(std::string const &fileName, std::string const &searchPath)
            :
                d_hasPath(true)  // overrule the interface-specified
                                // value
        {}
};
```

Here d_hasPath receives its value only once: it's always initialized to false except when the shown constructor is used in which case it is initialized to true.

7.5 Uniform initialization

When defining variables and objects they may immediately be given initial values. Class type objects are always initialized using one of their available constructors. C already supports the array and struct initializer list consisting of a list of constant expressions surrounded by a pair of curly braces.
A comparable initialization, called uniform initialization is added to C++ by the C++11 standard. It uses the following syntax:

```cpp
Type object {value list};
```

When defining objects using a list of objects each individual object may use its own uniform initialization.

The advantage of uniform initialization over using constructors is that using constructor arguments may sometimes result in an ambiguity as constructing an object may sometimes be confused with using the object’s overloaded function call operator (cf. section 11.10). As initializer lists can only be used with plain old data (POD) types (cf. section 9.9) and with classes that are ‘initializer list aware’ (like std::vector) the ambiguity does not arise when initializer lists are used.

Uniform initialization can be used to initialize an object or variable, but also to initialize data members in a constructor or implicitly in the return statement of functions. Examples:

```cpp
class Person
{
   // data members
   public:
      Person(std::string const &name, size_t mass)
      :
         d_name (name),
         d_mass (mass)
      {}
      Person copy() const
      {
         return {d_name, d_mass};
      }
};
```

Object definitions may be encountered in unexpected places, easily resulting in (human) confusion. Consider a function 'func' and a very simple class Fun (struct is used, as data hiding is not an issue here; in-class implementations are used for brevity):

```cpp
void func();

struct Fun
{
   Fun(void (*f)())
   {
      std::cout << "Constructor\n";
   };
   void process()
   {
      std::cout << "process\n";
   }
};
```

In main a Fun object is defined: Fun fun(func). Running this program displays Constructor: fun is constructed. Next we intend to call process for an anonymous Fun object:
Fun fun(func);
Fun(func).process();

Constructor appears twice, and then process is displayed.

What about just defining an anonymous Fun object? We do:

Fun(func);

Now we’re in for a surprise. The compiler complains that Fun’s default constructor is missing. Why’s that? Insert some blanks immediately after Fun and you get Fun(func). Parentheses around an identifier are OK, and are stripped off once the parenthesized expression has been parsed. In this case: (func) equals func, and so we have Fun(func); the definition of a Fun(func) object, using Fun’s default constructor (which isn’t provided).

So why does Fun(func).process() compile? In this case we have a member selector operator, whose left-hand operand must be an class-type object. The object must exist, and Fun(func) represents that object. It’s not the name of an existing object, but a constructor expecting a function like func exists. The compiler now creates an anonymous Fun, passing it func as its argument.

Clearly, in this example, parentheses cannot be used to create an anonymous Fun object. However, the uniform initialization can be used. To define the anonymous Fun object use this syntax:

Fun {func};

(which can also be used to immediately call one of its members. E.g., Fun(func).process()).

Although the uniform initialization syntax is slightly different from the syntax of an initializer list (the latter using the assignment operator) the compiler nevertheless uses the initializer list if a constructor supporting an initializer list is available. As an example consider:

class Vector
{
    public:
        Vector(size_t size);
        Vector(std::initializer_list<int> const &values);
    
    Vector vi = {4};

When defining vi the constructor expecting the initializer list is called rather than the constructor expecting a size_t argument. If the latter constructor is required the definition using the standard constructor syntax must be used. I.e., Vector vi(4).

Initializer lists are themselves objects that may be constructed using another initializer list. However, values stored in an initializer list are immutable. Once the initializer list has been defined their values remain as-is.

Before using the initializer_list the <initializer_list> header file must be included.

Initializer lists support a basic set of member functions and constructors:

- initializer_list<Type> object:
  defines object as an empty initializer list
• \texttt{initializer\_list<Type> object \{ list of Type values \}}: 
defines object as an initializer list containing Type values
• \texttt{initializer\_list<Type> object(other):} 
  initializes object using the values stored in other
• \texttt{size\_t size() const:} 
  returns the number of elements in the initializer list
• \texttt{Type const *begin() const:} 
  returns a pointer to the first element of the initializer list
• \texttt{Type const *end() const:} 
  returns a pointer just beyond the location of the last element of the initializer list

\section*{7.6 Defaulted and deleted class members}

In everyday class design two situations are frequently encountered:

\begin{itemize}
  \item A class offering constructors explicitly has to define a default constructor;
  \item A class (e.g., a class implementing a stream) cannot initialize objects by copying the values from an existing object of that class (called \textit{copy construction}) and cannot assign objects to each other.
\end{itemize}

Once a class defines at least one constructor its default constructor is not automatically defined by the compiler. The C++11 standard relaxes that restriction somewhat by offering the \texttt{= default} syntax. A class specifying \texttt{= default} with its default constructor declaration indicates that the trivial default constructor should be provided by the compiler. A trivial default constructor performs the following actions:

\begin{itemize}
  \item Its data members of built-in or primitive types are not initialized;
  \item Its composed (class type) data members are initialized by their default constructors.
  \item If the class is derived from a base class (cf. chapter 13) the base class is initialized by its default constructor.
\end{itemize}

Trivial implementations can also be provided for the \textit{copy constructor}, the \textit{overloaded assignment operator}, and the \textit{destructor}. Those members are introduced in chapter 9.

Conversely, situations exist where some (otherwise automatically provided) members should \textit{not} be made available. This is realized by specifying \texttt{= delete}. Using \texttt{= default} and \texttt{= delete} is illustrated by the following example. The default constructor receives its trivial implementation, copy-construction is prevented:

\begin{verbatim}
class Strings
{
  public:
  Strings() = default;
\end{verbatim}
Strings(std::string const *sp, size_t size);
Strings(Strings const &other) = delete;

7.7 Const member functions and const objects

The keyword const is often used behind the parameter list of member functions. This keyword indicates that a member function does not alter the data members of its object. Such member functions are called const member functions. In the class Person, we see that the accessor functions were declared const:

```cpp
class Person
{
  public:
    std::string const &name() const;
    std::string const &address() const;
    std::string const &phone() const;
    size_t mass() const;
};
```

The rule of thumb given in section 3.1.3 applies here too: whichever appears to the left of the keyword const, is not altered. With member functions this should be interpreted as ‘doesn’t alter its own data’.

When implementing a const member function the const attribute must be repeated:

```cpp
string const &Person::name() const
{
  return d_name;
}
```

The compiler prevents the data members of a class from being modified by one of its const member functions. Therefore a statement like

```cpp
d_name[0] = toupper(static_cast<unsigned char>(d_name[0]));
```

results in a compiler error when added to the above function’s definition.

Const member functions are used to prevent inadvertent data modification. Except for constructors and the destructor (cf. chapter 9) only const member functions can be used with (plain, references or pointers to) const objects.

Const objects are frequently encountered as const & parameters of functions. Inside such functions only the object’s const members may be used. Here is an example:

```cpp
void displayMass(ostream &out, Person const &person)
{
  out << person.name() << " weighs " << person.mass() << " kg.\n";
}
```
Since `person` is defined as a `Person const &` the function `displayMass` cannot call, e.g., `person.setMass(75)`.

The `const` member function attribute can be used to overload member functions. When functions are overloaded by their `const` attribute the compiler uses the member function matching most closely the `const`-qualification of the object:

- When the object is a `const` object, only `const` member functions can be used.
- When the object is not a `const` object, non-`const` member functions are used, unless only a `const` member function is available. In that case, the `const` member function is used.

The next example illustrates how (non) `const` member functions are selected:

```cpp
#include <iostream>
using namespace std;

class Members
{
    public:
        Members();
        void member();
        void member() const;
};

Members::Members()
{}
void Members::member()
{
    cout << "non const member\n";
}
void Members::member() const
{
    cout << "const member\n";
}

int main()
{
    Members const constObject;
    Members nonConstObject;

    constObject.member();
    nonConstObject.member();
}

/*
   Generated output:
   const member
   non const member
*/
```

As a general principle of design: member functions should always be given the `const` attribute, unless they actually modify the object's data.
7.7. CONST MEMBER FUNCTIONS AND CONST OBJECTS

7.7.1 Anonymous objects

Sometimes objects are used because they offer a certain functionality. The objects only exist because of their functionality, and nothing in the objects themselves is ever changed. The following class `Print` offers a facility to print a string, using a configurable prefix and suffix. A partial class interface could be:

```cpp
class Print {
public:
    Print(ostream &out);
    void print(std::string const &prefix, std::string const &text,
               std::string const &suffix) const;
};
```

An interface like this would allow us to do things like:

```cpp
Print print(cout);
for (int idx = 0; idx != argc; ++idx)
    print.print("arg: ", argv[idx], 
```

This works fine, but it could greatly be improved if we could pass `print`'s invariant arguments to `Print`'s constructor. This would simplify `print`'s prototype (only one argument would need to be passed rather than three) and we could wrap the above code in a function expecting a `Print` object:

```cpp
void allArgs(Print const &print, int argc, char *argv[])
{
    for (int idx = 0; idx != argc; ++idx)
        print.print(argv[idx]);
}
```

The above is a fairly generic piece of code, at least it is with respect to `Print`. Since `prefix` and `suffix` don't change they can be passed to the constructor which could be given the prototype:

```cpp
Print(ostream &out, string const &prefix = ", string const &suffix = ");
```

Now `allArgs` may be used as follows:

```cpp
Print p1(cout, "\", "); // prints to cout
Print p2(cerr, "err: --", "--\n"); // prints to cerr
allArgs(p1, argc, argv); // prints to cout
allArgs(p2, argc, argv); // prints to cerr
```

But now we note that `p1` and `p2` are only used inside the `allArgs` function. Furthermore, as we can see from `print`'s prototype, `print` doesn't modify the internal data of the `Print` object it is using.

In such situations it is actually not necessary to define objects before they are used. Instead anonymous objects may be used. Anonymous objects can be used:

- to initialize a function parameter which is a `const` reference to an object;
• if the object is only used inside the function call.

These anonymous objects are considered constant as they merely exist for passing the information of (class type) objects to functions. They are not considered ‘variables’. Of course, a `const_cast` could be used to cast away the const reference's constness, but any change is lost once the function returns. These anonymous objects used to initialize const references should not be confused with rvalue references (section 3.3.2) which have a completely different purpose in life. Rvalue references primarily exist to be ‘swallowed’ by functions receiving them. Thus, the information made available by rvalue references outlives the rvalue reference objects which are also anonymous.

Anonymous objects are defined when a constructor is used without providing a name for the constructed object. Here is the corresponding example:

```cpp
allArgs(Print(cout, "arg: ", "\n"), argc, argv);  // prints to cout
allArgs(Print(cerr, "err: --", "--\n"), argc, argv); // prints to cerr
```

In this situation the `Print` objects are constructed and immediately passed as first arguments to the `allArgs` functions, where they are accessible as the function's `print` parameter. While the `allArgs` function is executing they can be used, but once the function has completed, the anonymous `Print` objects are no longer accessible.

### 7.7.1.1 Subtleties with anonymous objects

Anonymous objects can be used to initialize function parameters that are `const` references to objects. These objects are created just before such a function is called, and are destroyed once the function has terminated. C++'s grammar allows us to use anonymous objects in other situations as well. Consider the following snippet of code:

```cpp
int main()
{
    // initial statements
    Print("hello", "world");
    // later statements
}
```

In this example an anonymous `Print` object is constructed, and it is immediately destroyed thereafter. So, following the 'initial statements' our `Print` object is constructed. Then it is destroyed again followed by the execution of the 'later statements'.

The example illustrates that the standard lifetime rules do not apply to anonymous objects. Their lifetimes are limited to the `statements`, rather than to the `end of the block` in which they are defined.

Plain anonymous object are at least useful in one situation. Assume we want to put markers in our code producing some output when the program's execution reaches a certain point. An object's constructor could be implemented so as to provide that marker-functionality allowing us to put markers in our code by defining anonymous, rather than named objects.

C++'s grammar contains another remarkable characteristic illustrated by the next example:

```cpp
int main(int argc, char **argv)
{
    Print p(cout, "", "");   // 1
    allArgs(Print(p), argc, argv); // 2
}
```
In this example a non-anonymous object \( p \) is constructed in statement 1, which is then used in statement 2 to initialize an anonymous object. The anonymous object, in turn, is then used to initialize allArg's const reference parameter. This use of an existing object to initialize another object is common practice, and is based on the existence of a so-called copy constructor. A copy constructor creates an object (as it is a constructor) using an existing object's characteristics to initialize the data of the object that's created. Copy constructors are discussed in depth in chapter 9, but presently only the concept of a copy constructor is used.

In the above example a copy constructor is used to initialize an anonymous object. The anonymous object was then used to initialize a parameter of a function. However, when we try to apply the same trick (i.e., using an existing object to initialize an anonymous object) to a plain statement, the compiler generates an error: the object \( p \) can't be redefined (in statement 3, below):

```c
int main(int argc, char *argv[]) {
    Print p("", ""); // 1
    allArgs(Print(p), argc, argv); // 2
    Print(p); // 3 error!
}
```

Does this mean that using an existing object to initialize an anonymous object that is used as function argument is OK, while an existing object can't be used to initialize an anonymous object in a plain statement?

The compiler actually provides us with the answer to this apparent contradiction. About statement 3 the compiler reports something like:

```
error: redeclaration of 'Print p'
```

which solves the problem when realizing that within a compound statement objects and variables may be defined. Inside a compound statement, a type name followed by a variable name is the grammatical form of a variable definition. Parentheses can be used to break priorities, but if there are no priorities to break, they have no effect, and are simply ignored by the compiler. In statement 3 the parentheses allowed us to get rid of the blank that's required between a type name and the variable name, but to the compiler we wrote

```
Print (p);
```

which is, since the parentheses are superfluous, equal to

```
Print p;
```

thus producing \( p \)'s redeclaration.

As a further example: when we define a variable using a built-in type (e.g., double) using superfluous parentheses the compiler quietly removes these parentheses for us:

```
double (((a)))); // weird, but OK.
```

To summarize our findings about anonymous variables:

- Anonymous objects are great for initializing const reference parameters.
• The same syntax, however, can also be used in stand-alone statements, in which they are interpreted as variable definitions if our intention actually was to initialize an anonymous object using an existing object.

• Since this may cause confusion, it’s probably best to restrict the use of anonymous objects to the first (and main) form: initializing function parameters.

### 7.8 Member function reference bindings (& and &&)

**TO DO**

### 7.9 The keyword ‘inline’

Let us take another look at the implementation of the function `Person::name()`:

```cpp
std::string const &Person::name() const
{
    return d_name;
}
```

This function is used to retrieve the name field of an object of the class `Person`. Example:

```cpp
void showName(Person const &person)
{
    cout << person.name();
}
```

To insert `person`’s name the following actions are performed:

• The function `Person::name()` is called.

• This function returns `person`’s `d_name` as a reference.

• The referenced name is inserted into `cout`.

Especially the first part of these actions causes some time loss, since an extra function call is necessary to retrieve the value of the `name` field. Sometimes a faster procedure immediately making the `d_name` data member available is preferred without ever actually calling a function `name`. This can be realized using **inline** functions. An inline function is a request to the compiler to insert the function’s code at the location of the function’s call. This may speed up execution by avoiding a function call, which typically comes with some (stack handling and parameter passing) overhead. Note that **inline** is a request to the compiler: the compiler may decide to ignore it, and **will** probably ignore it when the function’s body contains much code. Good programming discipline suggests to be aware of this, and to avoid **inline** unless the function’s body is fairly small. More on this in section 7.9.2.
7.9. THE KEYWORD ‘INLINE’

7.9.1 Defining members inline

Inline functions may be implemented in the class interface itself. For the class Person this results in the following implementation of name:

```cpp
class Person
{
    public:
        std::string const &name() const
        {
            return d_name;
        }
};
```

Note that the inline code of the function name now literally occurs inline in the interface of the class Person. The keyword const is again added to the function’s header.

Although members can be defined in-class (i.e., inside the class interface itself), it is considered bad practice for the following reasons:

- Defining members inside the interface contaminates the interface with implementations. The interface’s purpose is to document what functionality the class offers. Mixing member declarations and implementation details complicates understanding the interface. Readers need to skip implementation details which takes time and makes it hard to grab the ‘broad picture’, and thus to understand at a glance what functionality the class’s objects are offering.

- In-class implementations of private member functions may usually be avoided altogether (as they are private members). They should be moved to the internal header file (unless inline public members use such inline private members).

- Although members that are eligible for inline-coding should remain inline, situations do exist where such inline members migrate from an inline to a non-inline definition. In-class inline definitions still need editing (sometimes considerable editing) before they can be compiled. This additional editing is undesirable.

Because of the above considerations inline members should not be defined in-class. Rather, they should be defined following the class interface. The Person::name member is therefore preferably defined as follows:

```cpp
class Person
{
    public:
        std::string const &name() const;
};
```

```cpp
inline std::string const &Person::name() const
{
    return d_name;
}
```

If it is ever necessary to cancel Person::name’s inline implementation, then this becomes its non-inline implementation:

```cpp
#include "person.ih"
```
std::string const &Person::name() const
{
    return d_name;
}

Only the inline keyword needs to be removed to obtain the correct non-inline implementation.

Defining members inline has the following effect: whenever an inline-defined function is called, the compiler may insert the function’s body at the location of the function call. It may be that the function itself is never actually called.

This construction, where the function code itself is inserted rather than a call to the function, is called an inline function. Note that using inline functions may result in multiple occurrences of the code of those functions in a program: one copy for each invocation of the inline function. This is probably OK if the function is a small one, and needs to be executed fast. It’s not so desirable if the code of the function is extensive. The compiler knows this too, and handles the use of inline functions as a request rather than a command. If the compiler considers the function too long, it will not grant the request. Instead it will treat the function as a normal function.

7.9.2 When to use inline functions

When should inline functions be used, and when not? There are some rules of thumb which may be followed:

- In general inline functions should not be used. Voilà; that’s simple, isn’t it?
- Consider defining a function inline once a fully developed and tested program runs too slowly and shows ‘bottlenecks’ in certain functions, and the bottleneck is removed by defining inline members. A profiler, which runs a program and determines where most of the time is spent, is necessary to perform such optimizations.
- Defining inline functions may be considered when they consist of one very simple statement (such as the return statement in the function Person::name).
- When a function is defined inline, its implementation is inserted in the code wherever the function is used. As a consequence, when the implementation of the inline function changes, all sources using the inline function must be recompiled. In practice that means that all functions must be recompiled that include (either directly or indirectly) the header file of the class in which the inline function is defined. Not a very attractive prospect.
- It is only useful to implement an inline function when the time spent during a function call is long compared to the time spent by the function’s body. An example of an inline function which hardly affects the program’s speed is:

  inline void Person::printname() const
  {
      cout << d_name << ’\n’;
  }

  This function contains only one statement. However, the statement takes a relatively long time to execute. In general, functions which perform input and output take lots of time. The effect of the conversion of this function printname() to inline would therefore lead to an insignificant gain in execution time.
7.10. LOCAL CLASSES: CLASSES INSIDE FUNCTIONS

All inline functions have one disadvantage: the actual code is inserted by the compiler and must therefore be known at compile-time. Therefore, as mentioned earlier, an inline function can never be located in a run-time library. Practically this means that an inline function is found near the interface of a class, usually in the same header file. The result is a header file which not only shows the declaration of a class, but also part of its implementation, thus always blurring the distinction between interface and implementation.

7.9.2.1 A prelude: when NOT to use inline functions

As a prelude to chapter 14 (Polymorphism), there is one situation in which inline functions should definitely be avoided. At this point in the C++ Annotations it’s a bit too early to expose the full details, but since the keyword inline is the topic of this section this is considered the appropriate location for the advice.

There are situations where the compiler is confronted with so-called vague linkage (cf. http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Vague-Linkage.html). These situations occur when the compiler does not have a clear indication in what object file to put its compiled code. This happens, e.g., with inline functions, which are usually encountered in multiple source files. Since the compiler may insert the code of ordinary inline functions in places where these functions are called, vague linking is usually no problem with these ordinary functions.

However, as explained in chapter 14, when using polymorphism the compiler must ignore the inline keyword and define so-called virtual members as true (out-of-line functions). In this situation the vague linkage may cause problems, as the compiler must decide in what object s to put their code. Usually that’s not a big problem as long as the function is at least called once. But virtual functions are special in the sense that they may very well never be explicitly called. On some architectures (e.g., armeel) the compiler may fail to compile such inline virtual functions. This may result in missing symbols in programs using them. To make matters slightly more complex: the problem may emerge when shared libraries are used, but not when static libraries are used.

To avoid all of these problems virtual functions should never be defined inline, but they should always be defined out-of-line. I.e., they should be defined in source files.

7.10 Local classes: classes inside functions

Classes are usually defined at the global or namespace level. However, it is entirely possible to define a local class, i.e., inside a function. Such classes are called local classes.

Local classes can be very useful in advanced applications involving inheritance or templates (cf. section 13.9). At this point in the C++ Annotations they have limited use, although their main features can be described. At the end of this section an example is provided.

- Local classes may use almost all characteristics of normal classes. They may have constructors, destructors, data members, and member functions;
- Local classes cannot define static data members. Static member functions, however, can be defined.
- Since a local class may define static member functions, it is possible to define nested functions in C++ somewhat comparable to the way programming languages like Pascal allow nested functions to be defined.
• If a local class needs access to a constant integral value, a local \texttt{enum} can be used. The \texttt{enum} may be anonymous, exposing only the \texttt{enum} values.

• Local classes cannot directly access the non-static variables of their surrounding context. For example, in the example shown below the class \texttt{Local} cannot directly access \texttt{main}'s \texttt{argc} parameter.

• Local classes may directly access global data and static variables defined by their surrounding function. This includes variables defined in the anonymous namespace of the source file containing the local class.

• Local class objects can be defined inside the function body, but they cannot leave the function as objects of their own type. I.e., a local class name cannot be used for either the return type or for the parameter types of its surrounding function.

• As a prelude to \textit{inheritance} (chapter 13): a local class may be derived from an existing class allowing the surrounding function to return a dynamically allocated locally constructed class object, pointer or reference could be returned via a base class pointer or reference.

```cpp
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char *argv[]) {
    static size_t staticValue = 0;
    class Local {
        int d_argc; // non-static data members OK
        public:
            enum // enums OK
            {
                VALUE = 5
            };
            Local(int argc) // constructors and member functions OK
            : d_argc(argc) // in-class implementation required
            {
                // global data: accessible
                cout << "Local constructor\n";
                // static function variables: accessible
                staticValue += 5;
            }
            static void hello() // static member functions: OK
            {
                cout << "hello world\n";
            }
    };
    Local::hello(); // call Local static member
    Local loc(argc); // define object of a local class.
}
```
7.11 The keyword ‘mutable’

Earlier, in section 7.7, the concepts of const member functions and const objects were introduced.

C++ also allows the declaration of data members which may be modified, even by const member function. The declaration of such data members in the class interface start with the keyword `mutable`.

Mutable should be used for those data members that may be modified without logically changing the object, which might therefore still be considered a constant object.

An example of a situation where `mutable` is appropriately used is found in the implementation of a string class. Consider the `std::string`'s `c_str` and `data` members. The actual data returned by the two members are identical, but `c_str` must ensure that the returned string is terminated by an 0-byte. As a string object has both a length and a capacity an easy way to implement `c_str` is to ensure that the string's capacity exceeds its length by at least one character. This invariant allows `c_str` to be implemented as follows:

```cpp
char const *string::c_str() const
{
    d_data[d_length] = 0;
    return d_data;
}
```

This implementation logically does not modify the object’s data as the bytes beyond the object’s initial (length) characters have undefined values. But in order to use this implementation `d_data` must be declared `mutable`:

```cpp
mutable char *d_data;
```

The keyword `mutable` is also useful in classes implementing, e.g., reference counting. Consider a class implementing reference counting for textstrings. The object doing the reference counting might be a const object, but the class may define a copy constructor. Since const objects can’t be modified, how would the copy constructor be able to increment the reference count? Here the `mutable` keyword may profitably be used, as it can be incremented and decremented, even though its object is a const object.

The keyword `mutable` should sparingly be used. Data modified by const member functions should never logically modify the object, and it should be easy to demonstrate this. As a rule of thumb: do not use `mutable` unless there is a very clear reason (the object is logically not altered) for violating this rule.

7.12 Header file organization

In section 2.5.10 the requirements for header files when a C++ program also uses C functions were discussed. Header files containing class interfaces have additional requirements.

First, source files. With the exception of the occasional classless function, source files contain the code of member functions of classes. here there are basically two approaches:

- All required header files for a member function are included in each individual source file.
- All required header files (for all member functions of a class) are included in a header file that is included by each of the source files defining class members.
CHAPTER 7. CLASSES

The first alternative has the advantage of economy for the compiler: it only needs to read the header files that are necessary for a particular source file. It has the disadvantage that the program developer must include multiple header files again and again in source files: it both takes time to type the include-directives and to think about the header files which are needed in a particular source file.

The second alternative has the advantage of economy for the program developer: the header file of the class accumulates header files, so it tends to become more and more generally useful. It has the disadvantage that the compiler frequently has to process many header files which aren’t actually used by the function to compile.

With computers running faster and faster (and compilers getting smarter and smarter) I think the second alternative is to be preferred over the first alternative. So, as a starting point source files of a particular class MyClass could be organized according to the following example:

```cpp
#include <myclass.h>

int MyClass::aMemberFunction()
{
}
```

There is only one include-directive. Note that the directive refers to a header file in a directory mentioned in the INCLUDE-file environment variable. Local header files (using #include "myclass.h") could be used too, but that tends to complicate the organization of the class header file itself somewhat.

The organization of the header file itself requires some attention. Consider the following example, in which two classes File and String are used.

Assume the File class has a member gets(String &destination), while the class String has a member function getline(File &file). The (partial) header file for the class String is then:

```cpp
#ifndef STRING_H_
#define STRING_H_

#include <project/file.h> // to know about a File

class String
{
    public:
        void getline(File &file);
};
#endif
```

Unfortunately a similar setup is required for the class File:

```cpp
#ifndef FILE_H_
#define FILE_H_

#include <project/string.h> // to know about a String

class File
{
    public:
        void gets(String &string);
};
```
Now we have created a problem. The compiler, trying to compile the source file of the function `File::gets` proceeds as follows:

- The header file `project/file.h` is opened to be read;
- `FILE_H_` is defined
- The header file `project/string.h` is opened to be read
- `STRING_H_` is defined
- The header file `project/file.h` is (again) opened to be read
- Apparently, `FILE_H_` is already defined, so the remainder of `project/file.h` is skipped.
- The interface of the class `String` is now parsed.
- In the class interface a reference to a `File` object is encountered.
- As the class `File` hasn’t been parsed yet, a `File` is still an undefined type, and the compiler quits with an error.

The solution to this problem is to use a *forward class reference before* the class interface, and to include the corresponding class header file *beyond* the class interface. So we get:

```cpp
#ifndef STRING_H_
#define STRING_H_

class File; // forward reference

class String
{
    public:
        void getLine(File &file);
};

#include <project/file.h> // to know about a File

#endif
```

A similar setup is required for the class `File`:

```cpp
#ifndef FILE_H_
#define FILE_H_

class String; // forward reference

class File
{
    public:
        void gets(String &string);
};
```

This works well in all situations where either references or pointers to other classes are involved and with (non-inline) member functions having class-type return values or parameters.

This setup doesn't work with composition, nor with in-class inline member functions. Assume the class `File` has a composed data member of the class `String`. In that case, the class interface of the class `File` must include the header file of the class `String` before the class interface itself, because otherwise the compiler can't tell how big a `File` object is. A `File` object contains a `String` member, but the compiler can't determine the size of that `String` data member and thus, by implication, it can't determine the size of a `File` object.

In cases where classes contain composed objects (or are derived from other classes, see chapter 13) the header files of the classes of the composed objects must have been read before the class interface itself. In such a case the class `File` might be defined as follows:

```cpp
#ifndef FILE_H_
#define FILE_H_

#include <project/string.h> // to know about a String

class File
{
    String d_line;          // composition!

    public:
        void gets(String &string);
};
#endif
```

The class `String` can't declare a `File` object as a composed member: such a situation would again result in an undefined class while compiling the sources of these classes.

All remaining header files (appearing below the class interface itself) are required only because they are used by the class's source files.

This approach allows us to introduce yet another refinement:

- Header files defining a class interface should declare what can be declared before defining the class interface itself. So, classes that are mentioned in a class interface should be specified using forward declarations unless
  - They are a base class of the current class (see chapter 13);
  - They are the class types of composed data members;
  - They are used in inline member functions.

In particular: additional actual header files are not required for:
  - class-type return values of functions;
  - class-type value parameters of functions.
Class header files of objects that are either composed or inherited or that are used in inline functions, must be known to the compiler before the interface of the current class starts. The information in the header file itself is protected by the `#ifndef ... #endif` construction introduced in section 2.5.10.

- Program sources in which the class is used only need to include this header file. Lakos, (2001) refines this process even further. See his book Large-Scale C++ Software Design for further details. This header file should be made available in a well-known location, such as a directory or subdirectory of the standard INCLUDE path.

- To implement member functions the class's header file is required and usually additional header files (like the string header file) as well. The class header file itself as well as these additional header files should be included in a separate internal header file (for which the extension .ih (‘internal header’) is suggested). The .ih file should be defined in the same directory as the source files of the class. It has the following characteristics:

  - There is no need for a protective `#ifndef .. #endif` shield, as the header file is never included by other header files.
  - The standard .h header file defining the class interface is included.
  - The header files of all classes used as forward references in the standard .h header file are included.
  - Finally, all other header files that are required in the source files of the class are included.

An example of such a header file organization is:

- First part, e.g., /usr/local/include/myheaders/file.h:

  ```
  #ifndef FILE_H_
  #define FILE_H_
  #include <fstream> // for composed ‘ifstream’
  class Buffer; // forward reference
  class File // class interface
  {
    std::ifstream d_instream;
    public:
      void gets(Buffer &buffer);
  }
  #endif
  #endif
  ```

- Second part, e.g., ~/myproject/file/file.ih, where all sources of the class File are stored:

  ```
  #include <myheaders/file.h> // make the class File known
  #include <buffer.h> // make Buffer known to File
  #include <string> // used by members of the class
  #include <sys/stat.h> // File.
7.12.1 Using namespaces in header files

When entities from namespaces are used in header files, no using directive should be specified in those header files if they are to be used as general header files declaring classes or other entities from a library. When the using directive is used in a header file then users of such a header file are forced to accept and use the declarations in all code that includes the particular header file.

For example, if in a namespace special an object Inserter cout is declared, then special::cout is of course a different object than std::cout. Now, if a class Flaw is constructed, in which the constructor expects a reference to a special::Inserter, then the class should be constructed as follows:

```cpp
class special::Inserter;

class Flaw
{
  public:
    Flaw(special::Inserter &ins);
};
```

Now the person designing the class Flaw may be in a lazy mood, and might get bored by continuously having to prefix special:: before every entity from that namespace. So, the following construction is used:

```cpp
using namespace special;

class Inserter;
class Flaw
{
  public:
    Flaw(Inserter &ins);
};
```

This works fine, up to the point where somebody wants to include flaw.h in other source files: because of the using directive, this latter person is now by implication also using namespace special, which could produce unwanted or unexpected effects:

```cpp
#include <flaw.h>
#include <iostream>
using std::cout;

int main()
{
  cout << "starting\n"; // won’t compile
}
```

The compiler is confronted with two interpretations for cout: first, because of the using directive in the flaw.h header file, it considers cout a special::Inserter, then, because of the using directive in the user program, it considers cout a std::ostream. Consequently, the compiler reports an error.

As a rule of thumb, header files intended for general use should not contain using declarations. This rule does not hold true for header files which are only included by the sources of a class: here
the programmer is free to apply as many using declarations as desired, as these directives never reach other sources.

### 7.13 Sizeof applied to class data members

In the C++11 standard the `sizeof` operator can be applied to data members of classes without the need to specify an object as well. Consider:

```cpp
class Data
{
    std::string d_name;
    ...  
};
```

To obtain the size of `Data`'s `d_name` member C++11 allows the following expression:

```cpp
sizeof(Data::d_name);
```

However, note that the compiler observes data protection here as well. `sizeof(Data::d_name)` can only be used where `d_name` may be visible as well, i.e., by `Data`'s member functions and friends.
Chapter 8

Static Data And Functions

In the previous chapters we provided examples of classes where each object had its own set of data members data. Each of the class's member functions could access any member of any object of its class.

In some situations it may be desirable to define common data fields, that may be accessed by all objects of the class. For example, the name of the startup directory, used by a program that recursively scans the directory tree of a disk. A second example is a variable that indicates whether some specific initialization has occurred. In that case the object that was constructed first would perform the initialization and would set the flag to 'done'.

Such situations are also encountered in C, where several functions need to access the same variable. A common solution in C is to define all these functions in one source file and to define the variable static: the variable name is invisible outside the scope of the source file. This approach is quite valid, but violates our philosophy of using only one function per source file. Another C-solution is to give the variable in question an unusual name, e.g., _6uldv8, hoping that other program parts won't use this name by accident. Neither the first, nor the second legacy C solution is elegant.

C++ solves the problem by defining static members: data and functions, common to all objects of a class and (when defined in the private section) inaccessible outside of the class. These static members are this chapter's topic.

Static members cannot be defined as virtual functions. A virtual member function is an ordinary member in that it has a this pointer. As static member functions have no this pointer, they cannot be declared virtual.

8.1 Static data

Any data member of a class can be declared static; be it in the public or private section of the class interface. Such a data member is created and initialized only once, in contrast to non-static data members which are created again and again for each object of the class.

Static data members are created as soon as the program starts. Even though they're created at the very beginning of a program's execution cycle they are nevertheless true members of their classes.

It is suggested to prefix the names of static member with s_ so they may easily be distinguished (in class member functions) from the class's data members (which should preferably start with d_).
Public static data members are global variable global variables. They may be accessed by all of the program’s code, simply by using their class names, the scope resolution operator and their member names. Example:

```cpp
class Test
{
    static int s_private_int;
    public:
        static int s_public_int;
};

int main()
{
    Test::s_public_int = 145; // OK
    Test::s_private_int = 12; // wrong, don’t touch
                             // the private parts
}
```

The example does not present an executable program. It merely illustrates the interface, and not the implementation of static data members, which is discussed next.

### 8.1.1 Private static data

To illustrate the use of a static data member which is a private variable in a class, consider the following:

```cpp
class Directory
{
    static char s_path[];
    public:
        // constructors, destructors, etc.
};
```

The data member `s_path[]` is a private static data member. During the program’s execution only one `Directory::s_path[]` exists, even though multiple objects of the class `Directory` may exist. This data member could be inspected or altered by the constructor, destructor or by any other member function of the class `Directory`.

Since constructors are called for each new object of a class, static data members are not initialized by constructors. At most they are modified. The reason for this is that static data members exist before any constructor of the class has been called. Static data members are initialized when they are defined, outside of any member function, exactly like the initialization of ordinary (non-class) global variables.

The definition and initialization of a static data member usually occurs in one of the source files of the class functions, preferably in a source file dedicated to the definition of static data members, called `data.cc`.

The data member `s_path[]`, used above, could thus be defined and initialized as follows in a file `data.cc`:
include "directory.ih"

char Directory::s_path[200] = "/usr/local";

In the class interface the static member is actually only declared. In its implementation (definition) its type and class name are explicitly mentioned. Note also that the size specification can be left out of the interface, as shown above. However, its size is (either explicitly or implicitly) required when it is defined.

Note that any source file could contain the definition of the static data members of a class. A separate data.cc source file is advised, but the source file containing, e.g., main() could be used as well. Of course, any source file defining static data of a class must also include the header file of that class, in order for the static data member to be known to the compiler.

A second example of a useful private static data member is given below. Assume that a class Graphics defines the communication of a program with a graphics-capable device (e.g., a VGA screen). The initialization of the device, which in this case would be to switch from text mode to graphics mode, is an action of the constructor and depends on a static flag variable s_nobjects. The variable s_nobjects simply counts the number of Graphics objects which are present at one time. Similarly, the destructor of the class may switch back from graphics mode to text mode when the last Graphics object ceases to exist. The class interface for this Graphics class might be:

class Graphics
{
    static int s_nobjects; // counts # of objects

    public:
        Graphics();
        ~Graphics(); // other members not shown.

    private:
        void setgraphicsmode(); // switch to graphics mode
        void settextmode(); // switch to text-mode
}

The purpose of the variable s_nobjects is to count the number of objects existing at a particular moment in time. When the first object is created, the graphics device is initialized. At the destruction of the last Graphics object, the switch from graphics mode to text mode is made:

    int Graphics::s_nobjects = 0; // the static data member

    Graphics::Graphics()
    {
        if (!s_nobjects++)
            setgraphicsmode();
    }

    Graphics::~Graphics()
    {
        if (!--s_nobjects)
            settextmode();
    }

Obviously, when the class Graphics would define more than one constructor, each constructor would need to increase the variable s_nobjects and would possibly have to initialize the graphics mode.
8.1.2 Public static data

Data members could also be declared in the public section of a class. This, however, is deprecated (as it violates the principle of data hiding). The static data member `s_path[]` (cf. section 8.1) could be declared in the public section of the class definition. This would allow all the program’s code to access this variable directly:

```cpp
int main()
{
    getcwd(Directory::s_path, 199);
}
```

A declaration is not a definition. Consequently the variable `s_path` still has to be defined. This implies that some source file still needs to contain `s_path[]` array’s definition.

8.1.3 Initializing static const data

Static `const` data members should be initialized like any other static data member: in source files defining these data members.

Usually, if these data members are of integral or built-in primitive data types the compiler accepts in-class initializations of such data members. However, there is no formal rule requiring the compiler to do so. Compilations may or may not succeed depending on the optimizations used by the compiler (e.g., using `-O2` may result in a successful compilation, but `-O0` (no-optimizations) may fail to compile, but then maybe only when shared libraries are used...).

In-class initializations of integer constant values (e.g., of types `char`, `int`, `long`, etc, maybe `unsigned`) is nevertheless possible using (e.g., anonymous) enums. The following example illustrates how this can be done:

```cpp
class X
{
    public:
        enum { s_x = 34 }; // In-class initialization of char
        enum: size_t { s_maxWidth = 100 }; // In-class initialization of unsigned size_t
};
```

To avoid confusion caused by different compiler options static data members should always explicitly be defined and initialized in a source file, whether or not `const`.

8.1.4 Generalized constant expressions (constexpr)

In C macros are often used to let the preprocessor perform simple calculations. These `macro functions` may have arguments, as illustrated in the next example:

```cpp
#define xabs(x) ((x) < 0 ? -(x) : (x))
```

The disadvantages of macros are well-known. The main reason for avoiding macros is that they are not parsed by the compiler, but are processed by the preprocessor resulting in mere text replacements and thus avoid type-safety or syntactic checks of the macro definition by itself. Furthermore,
since macros are processed by the preprocessor their use is unconditional, without acknowledging the context in which they are applied. NULL is an infamous example. Ever tried to define an enum symbol NULL? or EOF? Chances are that, if you did, the compiler threw strange error messages at you.

Generalized const expressions can be used as an alternative.

Generalized const expressions are recognized by the modifier constexpr (a keyword), that is applied to the expression’s type.

There is a small syntactic difference between the use of the const modifier and the use of the constexpr modifier. While the const modifier can be applied to definitions and declarations alike, the constexpr modifier can only be applied to definitions:

    extern int const externInt;    // OK: declaration of const int
    extern int constexpr error;    // ERROR: not a definition

Variables defined with the constexpr modifier have constant (immutable) values. But generalized const expressions are not just used to define constant variables; they have other applications as well. The constexpr keyword is usually applied to functions, turning the function into a constant-expression function.

A constant-expression function should not be confused with a function returning a const value (although a constant-expression function does return a (const) value). A constant expression function has the following characteristics:

- it returns a value;
- its return type is given the constexpr modifier;
- its body consists of one single return statement

Such functions are also called named constant expressions with parameters.

These constant expression functions may or may not be called with arguments that have been evaluated at compile-time (not just ‘const arguments’, as a const parameter value is not evaluated at compile-time). If they are called with compile-time evaluated arguments then the returned value is considered a const value as well.

This allows us to encapsulate expressions that can be evaluated at compile-time in functions, and it allows us to use these functions in situations where previously the expressions themselves had to be used. The encapsulation reduces the number of occurrences of the expressions to one, simplifying maintenance and reduces the probability of errors.

If arguments that could not be compile-time evaluated are passed to constant-expression functions, then these functions act like any other function, in that their return values are no longer considered constant expressions.

Assume some two-dimensional arrays must be converted to one-dimensional arrays. The one-dimensional array must have $nrows * ncols + nrows + ncols + 1$ elements, to store row, column, and total marginals, as well as the elements of the source array itself. Furthermore assume that nrows and ncols have been defined as globally available size_t const values (they could be a class’s static data). The one-dimensional arrays are data members of a class or struct, or they are also defined as global arrays.

Now that constant-expression functions are available the expression returning the number of the required elements can be encapsulated in such a function:
size_t const nRows = 45;
size_t const nCols = 10;

size_t constexpr nElements(size_t rows, size_t cols)
{
    return rows * cols + rows + cols + 1;
}

....

int intLinear[ nElements(nRows, nCols) ];

struct Linear
{
    double d_linear[ nElements(nRows, nCols) ];
};

If another part of the program needs to use a linear array for an array of different sizes then the constant-expression function can also be used. E.g.,

string stringLinear[ nElements(10, 4) ];

Constant-expression functions can be used in other constant expression functions as well. The following constant-expression function returns half the value, rounded upwards, that is returned by nElements:

size_t constexpr halfNElements(size_t rows, size_t cols)
{
    return (nElements(rows, cols) + 1) >> 1;
}

Classes should not expose their data members to external software, so as to reduce coupling between classes and external software. But if a class defines a static const size_t data member then that member’s value could very well be used to define entities living outside of the class’s scope, like the number of elements of an array or to define the value of some enum. In situations like these constant-expression functions are the perfect tool to maintain proper data hiding:

class Data
{
    static size_t const s_size = 7;

    public:
        static size_t constexpr size();
        size_t constexpr mSize();
};

size_t constexpr Data::size()
{
    return s_size;
}

size_t constexpr Data::mSize()
{
8.1. STATIC DATA

    return size();

    double data[ Data::size() ]; // OK: 7 elements
    short data2[ Data().mSize() ]; // also OK: see below

Please note the following:

- Constant-expression functions are implicitly declared inline;
- Non-static constant-expression member functions are implicitly const, and a const member modifier for them is optional;
- Constant values (e.g., static constant data members) used by constant-expression functions must be known by the time the compiler encounters the functions’ definitions. That’s why s_size was initialized in Data’s class interface.

8.1.4.1 Constant expression data

As we’ve seen, (member) functions and variables of primitive data types can defined with the constexpr modifier. What about class-type objects?

Objects of classes are values of class type, and like values of primitive types they can be defined with the constexpr specifier. Constant expression class-type objects must be initialized with constant expression arguments; the constructor that is actually used must itself have been declared with the constexpr modifier. Note again that the constexpr constructor’s definition must have been seen by the compiler before the constexpr object can be constructed:

    class ConstExpr
    {
        public:
            constexpr ConstExpr(int x);
    };

    ConstExpr ok(7); // OK: not declared as constexpr
    constexpr ConstExpr err(7); // ERROR: constructor’s definition not yet seen

    constexpr ConstExpr::ConstExpr(int x)
    {} 

    constexpr ConstExpr ok(7); // OK: definition seen
    constexpr ConstExpr okToo = ConstExpr(7); // also OK

A constant-expression constructor has the following characteristics:

- it is declared with the constexpr modifier;
- its member initializers only use constant expressions;
- its body is empty.
An object constructed with a constant-expression constructor is called a *user-defined literal*. De-
structors and copy constructors of user-defined literals must be trivial.

The `constexpr` characteristic of user-defined literals may or may not be maintained by its class’s members. If a member is not declared with a `constexpr` return value, then using that member does not result in a constant-expression. If a member *does* declare a `constexpr` return value then that member’s return value considered a `constexpr` if it is by itself a constant expression function. To maintain its `constexpr` characteristics it can refer to its classes data members *only* if its object has been defined with the `constexpr` modifier, as illustrated by the example:

```cpp
class Data
{
    int d_x;

    public:
        constexpr Data(int x) :
            d_x(x)
        {}

        int constexpr cMember()
        {
            return d_x;
        }

        int member() const
        {
            return d_x;
        }
};

Data d1(0); // OK, but not a constant expression

enum e1 {
    ERR = d1.cMember() // ERROR: cMember(): no constant
}; // expression anymore

constexpr Data d2(0); // OK, constant expression

enum e2 {
    OK = d2.cMember(), // OK: cMember(): now a constant
    // expression
    ERR = d2.member(), // ERR: member(): not a constant
}; // expression
```

### 8.2 Static member functions

In addition to static data members, C++ allows us to define *static member functions*. Similar to static data that are shared by all objects of the class, static member functions also exist without any associated object of their class.

Static member functions can access all static members of their class, but *also* the members (private or public) of objects of their class *if* they are informed about the existence of these objects (as in
the upcoming example). As static member functions are not associated with any object of their class they do not have a this pointer. In fact, a static member function is completely comparable to a global function, not associated with any class (i.e., in practice they are. See the next section (8.2.1) for a subtle note). Since static member functions do not require an associated object, static member functions declared in the public section of a class interface may be called without specifying an object of its class. The following example illustrates this characteristic of static member functions:

```cpp
class Directory
{
    string d_currentPath;
    static char s_path[];

    public:
        static void setpath(char const *newpath);
        static void preset(Directory &dir, char const *newpath);
    
    inline void Directory::preset(Directory &dir, char const *newpath)
    {
        // see the text below
        dir.d_currentPath = newpath; // 1
    }

    char Directory::s_path[200] = "/usr/local"; // 2

    void Directory::setpath(char const *newpath)
    {
        if (strlen(newpath) >= 200)
            throw "newpath too long";

        strcpy(s_path, newpath); // 3
    }

    int main()
    {
        Directory dir;

        Directory::setpath("/etc"); // 4
        dir.setpath("/etc"); // 5

        Directory::preset(dir, "/usr/local/bin"); // 6
        dir.preset(dir, "/usr/local/bin"); // 7
    }
```

- at 1 a static member function modifies a private data member of an object. However, the object whose member must be modified is given to the member function as a reference parameter. Note that static member functions can be defined as inline functions.
- at 2 a relatively long array is defined to be able to accommodate long paths. Alternatively, a string or a pointer to dynamic memory could be used.
- at 3 a (possibly longer, but not too long) new pathname is stored in the static data member s_path[]. Note that only static members are used.
- at 4, setpath() is called. It is a static member, so no object is required. But the compiler must
know to which class the function belongs, so the class is mentioned using the scope resolution operator.

- At 5, the same is implemented as in 4. Here `dir` is used to tell the compiler that we’re talking about a function in the `Directory` class. Static member functions can be called as normal member functions, but this does not imply that the static member function receives the object’s address as a `this` pointer. Here the member-call syntax is used as an alternative for the class name plus scope resolution operator syntax.

- At 6, `currentPath` is altered. As in 4, the class and the scope resolution operator are used.

- At 7, the same is implemented as in 6. But here `dir` is used to tell the compiler that we’re talking about a function in the `Directory` class. Here in particular note that this is not using `preset()` as an ordinary member function of `dir`: the function still has no `this`-pointer, so `dir` must be passed as argument to inform the static member function `preset` about the object whose `currentPath` member it should modify.

In the example only public static member functions were used. C++ also allows the definition of private static member functions. Such functions can only be called by member functions of their class.

### 8.2.1 Calling conventions

As noted in the previous section, static (public) member functions are comparable to classless functions. However, formally this statement is not true, as the C++ standard does not prescribe the same calling conventions for static member functions as for classless global functions.

In practice the calling conventions are identical, implying that the address of a static member function could be used as an argument of functions having parameters that are pointers to (global) functions.

If unpleasant surprises must be avoided at all cost, it is suggested to create global classless wrapper functions around static member functions that must be used as call back functions for other functions.

Recognizing that the traditional situations in which call back functions are used in C are tackled in C++ using template algorithms (cf. chapter 19), let’s assume that we have a class `Person` having data members representing the person’s name, address, phone and mass. Furthermore, assume we want to sort an array of pointers to `Person` objects, by comparing the `Person` objects these pointers point to. Keeping things simple, we assume that the following public static member exists:

```cpp
int Person::compare(Person const *const *p1, Person const *const *p2);
```

A useful characteristic of this member is that it may directly inspect the required data members of the two `Person` objects passed to the member function using pointers to pointers (double pointers).

Most compilers allow us to pass this function’s address as the address of the comparison function for the standard C `qsort()` function. E.g.,

```cpp
qsort
(
    personArray, nPersons, sizeof(Person *),
    reinterpret_cast<int(*)(void const *, void const *)>(Person::compare)
);
```
8.2. STATIC MEMBER FUNCTIONS

However, if the compiler uses different calling conventions for static members and for classless functions, this might not work. In such a case, a classless wrapper function like the following may be used profitably:

```cpp
int compareWrapper(void const *p1, void const *p2)
{
    return
    Person::compare
    (  
        static_cast<Person const *const *>(p1),
        static_cast<Person const *const *>(p2)
    );
}
```

resulting in the following call of the `qsort()` function:

```cpp
qsort(personArray, nPersons, sizeof(Person *), compareWrapper);
```

**Note:**

- The wrapper function takes care of any mismatch in the calling conventions of static member functions and classless functions;
- The wrapper function handles the required type casts;
- The wrapper function might perform small additional services (like dereferencing pointers if the static member function expects references to `Person` objects rather than double pointers);
- As an aside: in C++ programs functions like `qsort()`, requiring the specification of call back functions are seldom used. Instead using existing generic template algorithms is preferred (cf. chapter 19).
Chapter 9

Classes And Memory Allocation

In contrast to the set of functions that handle memory allocation in C (i.e., `malloc` etc.), memory allocation in C++ is handled by the operators `new` and `delete`. Important differences between `malloc` and `new` are:

- The function `malloc` doesn’t ‘know’ what the allocated memory will be used for. E.g., when memory for `int`s is allocated, the programmer must supply the correct expression using a multiplication by `sizeof(int)`. In contrast, `new` requires a type to be specified; the `sizeof` expression is implicitly handled by the compiler. Using `new` is therefore type safe.

- Memory allocated by `malloc` is initialized by `calloc`, initializing the allocated characters to a configurable initial value. This is not very useful when objects are available. As operator `new` knows about the type of the allocated entity it may (and will) call the constructor of an allocated class type object. This constructor may be also supplied with arguments.

- All C-allocation functions must be inspected for NULL-returns. This is not required anymore when `new` is used. In fact, `new`'s behavior when confronted with failing memory allocation is configurable through the use of a `new_handler` (cf. section 9.2.2).

A comparable relationship exists between `free` and `delete`: `delete` makes sure that when an object is deallocated, its destructor is automatically called.

The automatic calling of constructors and destructors when objects are created and destroyed has consequences which we shall discuss in this chapter. Many problems encountered during C program development are caused by incorrect memory allocation or memory leaks: memory is not allocated, not freed, not initialized, boundaries are overwritten, etc. C++ does not ‘magically’ solve these problems, but it does provide us with tools to prevent these kinds of problems.

As a consequence of `malloc` and friends becoming deprecated the very frequently used `str...` functions, like `strdup`, that are all `malloc` based, should be avoided in C++ programs. Instead, the facilities of the `string` class and operators `new` and `delete` should be used instead.

Memory allocation procedures influence the way classes dynamically allocating their own memory should be designed. Therefore, in this chapter these topics are discussed in addition to discussions about operators `new` and `delete`. We’ll first cover the peculiarities of operators `new` and `delete`, followed by a discussion about:

- the destructor: the member function that’s called when an object ceases to exist;
- the assignment operator, allowing us to assign an object to another object of its own class;


• the this pointer, allowing explicit references to the object for which a member function was called;
• the copy constructor: the constructor creating a copy of an object;
• the move constructor: a constructor creating an object from an anonymous temporary object.

9.1 Operators ‘new’ and ‘delete’

C++ defines two operators to allocate memory and to return it to the ‘common pool’. These operators are, respectively new and delete.

Here is a simple example illustrating their use. An int pointer variable points to memory allocated by operator new. This memory is later released by operator delete.

```cpp
int *ip = new int;
delete ip;
```

Here are some characteristics of operators new and delete:

- new and delete are operators and therefore do not require parentheses, as required for functions like malloc and free;
- new returns a pointer to the kind of memory that’s asked for by its operand (e.g., it returns a pointer to an int);
- new uses a type as its operand, which has the important benefit that the correct amount of memory, given the type of the object to be allocated, is made available;
- as a consequence, new is a type safe operator as it always returns a pointer to the type that was mentioned as its operand. In addition, the type of the receiving pointer must match the type specified with operator new;
- new may fail, but this is normally of no concern to the programmer. In particular, the program does not have to test the success of the memory allocation, as is required for malloc and friends. Section 9.2.2 delves into this aspect of new;
- delete returns void;
- for each call to new a matching delete should eventually be executed, lest a memory leak occurs;
- delete can safely operate on a 0-pointer (doing nothing);
- otherwise delete must only be used to return memory allocated by new. It should not be used to return memory allocated by malloc and friends.
- in C++ malloc and friends are deprecated and should be avoided.

Operator new can be used to allocate primitive types but also to allocate objects. When a primitive type or a struct type without a constructor is allocated the allocated memory is not guaranteed to be initialized to 0, but an initialization expression may be provided:

```cpp
int *v1 = new int;           // not guaranteed to be initialized to 0
int *v1 = new int();         // initialized to 0
int *v2 = new int(3);        // initialized to 3
int *v3 = new int(3 + *v2);  // initialized to 9
```
When a class-type object is allocated, the arguments of its constructor (if any) are specified immediately following the type specification in the `new` expression and the object is initialized by to the thus specified constructor. For example, to allocate `string` objects the following statements could be used:

```c++
string *s1 = new string; // uses the default constructor
string *s2 = new string(); // same
string *s3 = new string(4, ''); // initializes to 4 blanks.
```

In addition to using `new` to allocate memory for a single entity or an array of entities there is also a variant that allocates raw memory: `operator new(sizeInBytes)`. Raw memory is returned as a `void*`. Here `new` allocates a block of memory for unspecified purpose. Although raw memory may consist of multiple characters it should not be interpreted as an array of characters. Since raw memory returned by `new` is returned as a `void*` its return value can be assigned to a `void*` variable. More often it is assigned to a `char*` variable, using a cast. Here is an example:

```c++
char *chPtr = static_cast<char*>(operator new(numberOfBytes));
```

The use of raw memory is frequently encountered in combination with the `placement new` operator, discussed in section 9.1.5.

### 9.1.1 Allocating arrays

Operator `new[]` is used to allocate arrays. The generic notation `new[]` is used in the C++ Annotations. Actually, the number of elements to be allocated must be specified between the square brackets and it must, in turn, be prefixed by the type of the entities that must be allocated. Example:

```c++
int *intarr = new int[20]; // allocates 20 ints
string *stringarr = new string[10]; // allocates 10 strings.
```

Operator `new` is a different operator than operator `new[]`. A consequence of this difference is discussed in the next section (9.1.2).

Arrays allocated by operator `new[]` are called dynamic arrays. They are constructed during the execution of a program, and their lifetime may exceed the lifetime of the function in which they were created. Dynamically allocated arrays may last for as long as the program runs.

When `new[]` is used to allocate an array of primitive values or an array of objects, `new[]` must be specified with a type and an (unsigned) expression between its square brackets. The type and expression together are used by the compiler to determine the required size of the block of memory to make available. When `new[]` is used the array's elements are stored consecutively in memory. An array index expression may thereafter be used to access the array's individual elements; `intarr[0]` represents the first `int` value, immediately followed by `intarr[1]`, and so on until the last element (`intarr[19]`). With non-class types (primitive types, `struct` types without constructors) the block of memory returned by operator `new[]` is not guaranteed to be initialized to 0.

When operator `new[]` is used to allocate arrays of objects their constructors are automatically used. Consequently `new string[20]` results in a block of 20 initialized `string` objects. When allocating arrays of objects the class's `default constructor` is used to initialize each individual object in turn. A non-default constructor cannot be called, but often it is possible to work around that as discussed in section 13.9.
The expression between brackets of operator `new[]` represents the number of elements of the array to allocate. The C++ standard allows allocation of 0-sized arrays. The statement `new int[0]` is correct C++. However, it is also pointless and confusing and should be avoided. It is pointless as it doesn’t refer to any element at all, it is confusing as the returned pointer has a useless non-0 value. A pointer intending to point to an array of values should be initialized (like any pointer that isn’t yet pointing to memory) to 0, allowing for expressions like `if (ptr) ...`

Without using operator `new[]`, arrays of variable sizes can also be constructed as local arrays. Such arrays are not dynamic arrays and their lifetimes are restricted to the lifetime of the block in which they were defined.

Once allocated, all arrays have fixed sizes. There is no simple way to enlarge or shrink arrays. C++ has no operator ‘renew’. Section 9.1.3 illustrates how to enlarge arrays.

### 9.1.2 Deleting arrays

Dynamically allocated arrays are deleted using operator `delete[]`. It expects a pointer to a block of memory, previously allocated by operator `new[]`.

When operator `delete[]`’s operand is a pointer to an array of objects two actions are performed:

- First, the class’s destructor is called for each of the objects in the array. The destructor, as explained later in this chapter, performs all kinds of cleanup operations that are required by the time the object ceases to exist.
- Second, the memory pointed at by the pointer is returned to the common pool.

Here is an example showing how to allocate and delete an array of 10 string objects:

```cpp
std::string *sp = new std::string[10];
delete[] sp;
```

No special action is performed if a dynamically allocated array of primitive typed values is deleted. Following `int *it = new int[10]` the statement `delete[] it` simply returns the memory pointed at by it. Realize that, as a pointer is a primitive type, deleting a dynamically allocated array of pointers to objects does not result in the proper destruction of the objects the array’s elements point at. So, the following example results in a memory leak:

```cpp
string **sp = new string *[5];
for (size_t idx = 0; idx != 5; ++idx)
    sp[idx] = new string;
delete[] sp; // MEMORY LEAK!
```

In this example the only action performed by `delete[]` is to return an area the size of five pointers to strings to the common pool.

Here’s how the destruction in such cases should be performed:

- Call `delete` for each of the array’s elements;
- Delete the array itself
Example:

```c++
for (size_t idx = 0; idx != 5; ++idx)
    delete sp[idx];
delete[] sp;
```

One of the consequences is of course that by the time the memory is going to be returned not only the pointer must be available but also the number of elements it contains. This can easily be accomplished by storing pointer and number of elements in a simple class and then using an object of that class.

Operator `delete[]` is a different operator than operator `delete`. The rule of thumb is: if `new[]` was used, also use `delete[]`.

### 9.1.3 Enlarging arrays

Once allocated, all arrays have fixed sizes. There is no simple way to enlarge or shrink arrays. C++ has no `renew` operator. The basic steps to take when enlarging an array are the following:

- Allocate a new block of memory of larger size;
- Copy the old array contents to the new array;
- Delete the old array;
- Let the pointer to the array point to the newly allocated array.

Static and local arrays cannot be resized. Resizing is only possible for dynamically allocated arrays. Example:

```c++
#include <string>
using namespace std;

string *enlarge(string *old, unsigned oldsize, unsigned newsize)
{
    string *tmp = new string[newsize]; // allocate larger array
    for (size_t idx = 0; idx != oldsize; ++idx)
        tmp[idx] = old[idx]; // copy old to tmp
    delete[] old; // delete the old array
    return tmp; // return new array
}

int main()
{
    string *arr = new string[4]; // initially: array of 4 strings
    arr = enlarge(arr, 4, 6); // enlarge arr to 6 elements.
}
```

The procedure to enlarge shown in the example also has several drawbacks.

- The new array requires `newsize` constructors to be called;
• Having initialized the strings in the new array, oldsize of them are immediately reassigned to the corresponding values in the original array;

• All the objects in the old arrays are destroyed.

Depending on the context various solutions exist to improve the efficiency of this rather inefficient procedure. An array of pointers could be used (requiring only the pointers to be copied, no destruction, no superfluous initialization) or raw memory in combination with the placement new operator could be used (an array of objects remains available, no destruction, no superfluous construction).

### 9.1.4 Managing ‘raw’ memory

As we’ve seen operator new allocates the memory for an object and subsequently initializes that object by calling one of its constructors. Likewise, operator delete calls an object’s destructor and subsequently returns the memory allocated by operator new to the common pool.

In the next section we’ll encounter another use of new, allowing us to initialize objects in so-called raw memory: memory merely consisting of bytes that have been made available by either static or dynamic allocation.

Raw memory is made available by operator new(sizeInBytes). This should not be interpreted as an array of any kind but just a series of memory locations that were dynamically made available. operator new returns a void * so a (static) cast is required to use it as memory of some type. Here are two examples:

```cpp
// room for 5 ints
int *ip = static_cast<int*>(operator new(5 * sizeof(int)));
// room for 5 strings
string *sp = static_cast<string*>(operator new(5 * sizeof(string)));
```

As operator new has no concept of data types the size of the intended data type must be specified when allocating raw memory for a certain number of objects of an intended type. The use of operator new therefore somewhat resembles the use of malloc.

The counterpart of operator new is operator delete. Operator delete expects a void * (so a pointer to any type can be passed to it). The pointer is interpreted as a pointer to raw memory which is returned to the common pool without any further action. In particular, no destructors are called by operator delete. The use of operator delete therefore resembles the use of free.

To return the memory pointed at by the abovementioned variables ip and sp operator delete should be used:

```cpp
// delete raw memory allocated by operator new
operator delete(ip);
operator delete(sp);
```

### 9.1.5 The ‘placement new’ operator

A remarkable form of operator new is called the placement new operator. Before using placement new the <memory> header file must be included.

Placement new is passed an existing block of memory into which new initializes an object or value. The block of memory should be large enough to contain the object, but apart from that there are
no further requirements. It is easy to determine how much memory is used by an entity (object or variable) of type `Type`: the `sizeof` operator returns the number of bytes used by an `Type` entity.

Entities may of course dynamically allocate memory for their own use. Dynamically allocated memory, however, is not part of the entity’s memory ‘footprint’ but it is always made available externally to the entity itself. This is why `sizeof` returns the same value when applied to different `string` objects that return different length and capacity values.

The placement `new` operator uses the following syntax (using `Type` to indicate the used data type):

```
Type *new(void *memory) Type(arguments);
```

Here, `memory` is a block of memory of at least `sizeof(Type)` bytes and `Type(arguments)` is any constructor of the class `Type`.

The placement `new` operator is useful in situations where classes set aside memory to be used later. This is used, e.g., by `std::string` to change its capacity. Calling `string::reserve` may enlarge that capacity without making memory beyond the string’s length immediately available to the string object’s users. But the object itself may use its additional memory. E.g., when information is added to a `string` object it can draw memory from its capacity rather than performing a reallocation for each single character that is added to its contents.

Let’s apply that philosophy to a class `Strings` storing `std::string` objects. The class defines a `string` accessing the memory holding its `d_size` `string` objects as well as `d_capacity - d_size` reserved memory. Assuming that a default constructor initializes `d_capacity` to 1, doubling `d_capacity` whenever an additional `string` must be stored, the class must support the following essential operations:

- doubling its capacity when all its spare memory (e.g., made available by `reserve`) has been consumed;
- adding another `string` object
- properly deleting the installed strings and memory when a `Strings` object ceases to exist.

The private member `void Strings::reserve` is called when the current capacity must be enlarged to `d_capacity`. It operates as follows: First new, raw, memory is allocated (line 1). This memory is in no way initialized with strings. Then the available strings in the old memory are copied into the newly allocated raw memory using placement `new` (line 2). Next, the old memory is deleted (line 3).

```
void Strings::reserve()
{
    using std::string;

    string *newMemory = static_cast<string *>(
        operator new(d_capacity * sizeof(string)));
    for (size_t idx = 0; idx != d_size; ++idx)
        new (newMemory + idx) string(d_memory[idx]);
    destroy();
    d_memory = newMemory;
}
```

The member `append` adds another `string` object to a `Strings` object. A (public) member `reserve(request)` (enlarging `d_capacity` if necessary and if enlarged calling `reserve()`) en-
sures that the `String` object’s capacity is sufficient. Then placement `new` is used to install the latest string into the raw memory’s appropriate location:

```cpp
void Strings::append(std::string const &next)
{
    reserve(d_size + 1);
    new (d_memory + d_size) std::string(next);
    ++d_size;
}
```

At the end of the `String` object’s lifetime, and during enlarging operations all currently used dynamically allocated memory must be returned. This is made the responsibility of the member `destroy`, which is called by the class’s destructor and by `reserve()`. More about the destructor itself in the next section, but the implementation of the support member `destroy` is discussed below.

With placement `new` an interesting situation is encountered. Objects, possibly themselves allocating memory, are installed in memory that may or may not have been allocated dynamically, but that is usually not completely filled with such objects. So a simple `delete[]` can’t be used. On the other hand, a `delete` for each of the objects that are available can’t be used either, since those `delete` operations would also try to delete the memory of the objects themselves, which wasn’t dynamically allocated.

This peculiar situation is solved in a peculiar way, only encountered in cases where placement `new` is used: memory allocated by objects initialized using placement `new` is returned by explicitly calling the object’s destructor. The destructor is declared as a member having as its name the class name preceded by a tilde, not using any arguments. So, `std::string`’s destructor is named `~string`. An object’s destructor only returns memory allocated by the object itself and, despite of its name, does not destroy its object. Any memory allocated by the `strings` stored in our class `Strings` is therefore properly destroyed by explicitly calling their destructors. Following this `d_memory` is back to its initial status: it again points to raw memory. This raw memory is then returned to the common pool by `operator delete`:

```cpp
void Strings::destroy()
{
    for (std::string *sp = d_memory + d_size; sp-- != d_memory; )
        sp->~string();
    operator delete(d_memory);
}
```

So far, so good. All is well as long as we’re using but one object. What about allocating an array of objects? Initialization is performed as usual. But as with `delete, delete[]` cannot be called when the buffer was allocated statically. Instead, when multiple objects were initialized using placement `new` in combination with a statically allocated buffer all the objects’ destructors must be called explicitly, as in the following example:

```cpp
using std::string;

char buffer[3 * sizeof(string)];
string *sp = new(buffer) string [3];

for (size_t idx = 0; idx < 3; ++idx)
    sp[idx].~string();
```
9.2 The destructor

Comparable to the constructor, classes may define a destructor. This function is the constructor’s counterpart in the sense that it is invoked when an object ceases to exist. A destructor is usually called automatically, but that’s not always true. The destructors of dynamically allocated objects are not automatically activated, but in addition to that: when a program is interrupted by an exit call, only the destructors of already initialized global objects are called. In that situation destructors of objects defined locally by functions are also not called. This is one (good) reason for avoiding exit in C++ programs.

Destructors obey the following syntactical requirements:

- a destructor’s name is equal to its class name prefixed by a tilde;
- a destructor has no arguments;
- a destructor has no return value.

Destructors are declared in their class interfaces. Example:

```cpp
class Strings
{
    public:
        Strings();
        ~Strings(); // the destructor
};
```

By convention the constructors are declared first. The destructor is declared next, to be followed by other member functions.

A destructor’s main task is to ensure that memory allocated by an object is properly returned when the object ceases to exist. Consider the following interface of the class Strings:

```cpp
class Strings
{
    std::string *d_string;
    size_t d_size;

    public:
        Strings();
        Strings(char const *const *cStrings, size_t n);
        ~Strings();

        std::string const &at(size_t idx) const;
        size_t size() const;
};
```

The constructor’s task is to initialize the data fields of the object. E.g, its constructors are defined as follows:

```cpp
Strings::Strings()
:
    d_string(0),
```
d_size(0)
{
}

Strings::Strings(char const *const *cStrings, size_t size)
: d_string(new string[size]), d_size(size)
{
    for (size_t idx = 0; idx != size; ++idx)
        d_string[idx] = cStrings[idx];
}

As objects of the class Strings allocate memory a destructor is clearly required. Destructors may or may not be called automatically. Here are the rules:

- Destructors are only called for fully constructed objects. C++ considers the object to be fully constructed once at least one of its constructors has normally completed. It used to be the constructor, but as C++11 supports constructor delegation, multiple constructors can be activated for a single object; hence ‘at least one constructor’. The remaining rules only apply to fully constructed objects;
- Destructors of local non-static objects are called automatically when the execution flow leaves the block in which they were defined; the destructors of objects defined somewhere in the outer block of a function are called just before the function terminates.
- Destructors of static or global objects are called when the program itself terminates.
- The destructor of a dynamically allocated object is called by delete using the object’s address as its operand;
- The destructors of a dynamically allocated array of objects are called by delete[] using the address of the array’s first element as its operand;
- The destructor of an object initialized by placement new is activated by explicitly calling the object’s destructor.

The destructor’s task is to ensure that all memory that is dynamically allocated and controlled only by the object itself is returned. The task of the Strings’s destructor would therefore be to delete the memory to which d_string points. Its implementation is:

```cpp
Strings::~Strings()
{
    delete[] d_string;
}
```

The next example shows Strings at work. In process a Strings store is created, and its data are displayed. It returns a dynamically allocated Strings object to main. A Strings * receives the address of the allocated object and deletes the object again. Another Strings object is then created in a block of memory made available locally in main, and an explicit call to ~Strings is required to return the memory allocated by that object. In the example only once a Strings object is automatically destroyed: the local Strings object defined by process. The other two Strings objects require explicit actions to prevent memory leaks.

```cpp
#include "strings.h"
```
#include <iostream>

using namespace std;

void display(Strings const &store) {
    for (size_t idx = 0; idx != store.size(); ++idx)
        cout << store.at(idx) << 'n';
}

Strings *process(char *argv[], int argc) {
    Strings store(argv, argc);
    display(store);
    return new Strings(argv, argc);
}

int main(int argc, char*argv[]) {
    Strings *sp = process(argv, argc);
    delete sp;

    char buffer[sizeof(Strings)];
    sp = new (buffer) Strings(argv, argc);
    sp->~Strings();
}

9.2.1 Object pointers revisited

Operators new and delete are used when an object or variable is allocated. One of the advantages of the operators new and delete over functions like malloc and free is that new and delete call the corresponding object constructors and destructors.

The allocation of an object by operator new is a two-step process. First the memory for the object itself is allocated. Then its constructor is called, initializing the object. Analogously to the construction of an object, the destruction is also a two-step process: first, the destructor of the class is called deleting the memory controlled by the object. Then the memory used by the object itself is freed.

Dynamically allocated arrays of objects can also be handled by new and delete. When allocating an array of objects using operator new the default constructor is called for each object in the array. In cases like this operator delete[] must be used to ensure that the destructor is called for each of the objects in array.

However, the addresses returned by new Type and new Type[size] are of identical types, in both cases a Type *. Consequently it cannot be determined by the type of the pointer whether a pointer to dynamically allocated memory points to a single entity or to an array of entities.

What happens if delete rather than delete[] is used? Consider the following situation, in which the destructor ~Strings is modified so that it tells us that it is called. In a main function an array of two Strings objects is allocated by new, to be deleted by delete[]. Next, the same actions are repeated, albeit that the delete operator is called without []:

#include <iostream>
#include "strings.h"
using namespace std;

Strings::~Strings()
{
    cout << "Strings destructor called" << '\n';
}

int main()
{
    Strings *a = new Strings[2];
    cout << "Destruction with []'s" << '\n';
    delete[] a;
    a = new Strings[2];
    cout << "Destruction without []'s" << '\n';
    delete a;
}

/*
Generated output:
Destruction with []'s
Strings destructor called
Strings destructor called
Destruction without []'s
Strings destructor called
*/

From the generated output, we see that the destructors of the individual Strings objects are called when delete[] is used, while only the first object's destructor is called if the [] is omitted.

Conversely, if delete[] is called in a situation where delete should have been called the results are unpredictable, and the program will most likely crash. This problematic behavior is caused by the way the run-time system stores information about the size of the allocated array (usually right before the array's first element). If a single object is allocated the array-specific information is not available, but it is nevertheless assumed present by delete[]. Thus this latter operator encounters bogus values in the memory locations just before the array's first element. It then dutifully interprets the value it encounters there as size information, usually causing the program to fail.

If no destructor is defined, a trivial destructor is defined by the compiler. The trivial destructor ensures that the destructors of composed objects (as well as the destructors of base classes if a class is a derived class, cf. chapter 13) are called. This has serious implications: objects allocating memory create memory leaks unless precautionary measures are taken (by defining an appropriate destructor). Consider the following program:

#include <iostream>
#include "strings.h"
using namespace std;

Strings::~Strings()
{
    cout << "Strings destructor called" << '\n';
}

int main()
9.2. THE DESTRUCTOR

This program produces no output at all. Why is this? The variable `ptr` is defined as a pointer to a pointer. The dynamically allocated array therefore consists of pointer variables and pointers are of a primitive type. No destructors exist for primitive typed variables. Consequently only the array itself is returned, and no `Strings` destructor is called.

Of course, we don’t want this, but require the `Strings` objects pointed to by the elements of `ptr` to be deleted too. In this case we have two options:

- In a for-statement visit all the elements of the `ptr` array, calling `delete` for each of the array’s elements. This procedure was demonstrated in the previous section.

- A wrapper class is designed around a pointer (to, e.g., an object of some class, like `Strings`). Rather than using a pointer to a pointer to `Strings` objects a pointer to an array of wrapper-class objects is used. As a result `delete[] ptr` calls the destructor of each of the wrapper-class objects, in turn calling the `Strings` destructor for their `d_strings` members. Example:

```cpp
#include <iostream>
using namespace std;

class Strings // partially implemented
{
    public:
        ~Strings();
};

inline Strings::~Strings()
{
    cout << "destructor called\n";
}

class Wrapper
{
    Strings *d_strings;

    public:
        Wrapper();
        ~Wrapper();
};

inline Wrapper::Wrapper()
{
    d_strings(new Strings())
}
inline Wrapper::~Wrapper()
{
    delete d_strings;
}
```
int main()
{
    auto ptr = new Strings *[4];
    // ... code assigning 'new Strings' to ptr's elements
    delete[] ptr; // memory leak: ~Strings() not called

    cout << "==========\n";
    delete[] new Wrapper[4]; // OK: 4 x destructor called
}

/*
Generated output:
==========
destructor called
destructor called
destructor called
destructor called
*/

9.2.2 The function set_new_handler()

The C++ run-time system ensures that when memory allocation fails an error function is activated. By default this function throws a bad_alloc exception (see section 10.8), terminating the program. Therefore it is not necessary to check the return value of operator new. Operator new's default behavior may be modified in various ways. One way to modify its behavior is to redefine the function that's called when memory allocation fails. Such a function must comply with the following requirements:

- it has no parameters;
- its return type is void.

A redefined error function might, e.g., print a message and terminate the program. The user-written error function becomes part of the allocation system through the function set_new_handler.

Such an error function is illustrated below:

```cpp
#include <iostream>
#include <string>
#include <cstring>

using namespace std;

void outOfMemory()
{
    cout << "Memory exhausted. Program terminates.\n" << '\n';
    exit(1);
}
```

1 This implementation applies to the Gnu C/C++ requirements. Actually using the program given in the next example is not advised, as it probably enormously slows down your computer due to the resulting use of the operating system's swap area.
int main()
{
    long allocated = 0;

    set_new_handler(outOfMemory); // install error function

    while (true) // eat up all memory
    {
        memset(new int [100000], 0, 100000 * sizeof(int));
        allocated += 100000 * sizeof(int);
        cout << "Allocated " << allocated << " bytes\n";
    }
}

Once the new error function has been installed it is automatically invoked when memory allocation fails, and the program is terminated. Memory allocation may fail in indirectly called code as well, e.g., when constructing or using streams or when strings are duplicated by low-level functions.

So far for the theory. On some systems the ‘out of memory’ condition may actually never be reached, as the operating system may interixe before the run-time support system gets a chance to stop the program (see also this link\(^2\)).

The standard C functions allocating memory (like `strdup`, `malloc`, `realloc` etc.) do not trigger the `new` handler when memory allocation fails and should be avoided in C++ programs.

### 9.3 The assignment operator

In C++ struct and class type objects can be directly assigned new values in the same way as this is possible in C. The default action of such an assignment for non-class type data members is a straight byte-by-byte copy from one data member to another. For now we’ll use the following simple class `Person`:

```cpp
class Person
{
    char *d_name;
    char *d_address;
    char *d_phone;

public:
    Person();
    Person(char const *name, char const *addr, char const *phone);
    ~Person();
private:
    char *strdupnew(char const *src); // returns a copy of src.
};
```

// strdupnew is easily implemented, here is its inline implementation:
inline char *Person::strdupnew(char const *src)
{
    return strcpy(new char [strlen(src) + 1], src);
}

Person's data members are initialized to zeroes or to copies of the NTB strings passed to Person's constructor, using some variant of strdup. The allocated memory is eventually returned by Person's destructor.

Now consider the consequences of using Person objects in the following example:

```cpp
void tmpPerson(Person const &person)
{
    Person tmp;
    tmp = person;
}
```

Here's what happens when tmpPerson is called:

- it expects a reference to a Person as its parameter `person`.
- it defines a local object `tmp`, whose data members are initialized to zeroes.
- the object referenced by `person` is copied to `tmp`: `sizeof(Person)` number of bytes are copied from `person` to `tmp`.

Now a potentially dangerous situation has been created. The actual values in `person` are pointers, pointing to allocated memory. After the assignment this memory is addressed by two objects: `person` and `tmp`.

- The potentially dangerous situation develops into an acutely dangerous situation once the function `tmpPerson` terminates: `tmp` is destroyed. The destructor of the class `Person` releases the memory pointed to by the fields `d_name`, `d_address` and `d_phone`: unfortunately, this memory is also pointed at by `person`.

This problematic assignment is illustrated in Figure 9.1.

Having executed `tmpPerson`, the object referenced by `person` now contains pointers to deleted memory.

This is undoubtedly not a desired effect of using a function like `tmpPerson`. The deleted memory is likely to be reused by subsequent allocations. The pointer members of `person` have effectively become wild pointers, as they don't point to allocated memory anymore. In general it can be concluded that

> every class containing pointer data members is a potential candidate for trouble.

Fortunately, it is possible to prevent these troubles, as discussed next.

### 9.3.1 Overloading the assignment operator

Obviously, the right way to assign one Person object to another, is not to copy the contents of the object bytewise. A better way is to make an equivalent object. One having its own allocated memory containing copies of the original strings.

The way to assign a Person object to another is illustrated in Figure 9.2. There are several ways to assign a Person object to another. One way would be to define a special member function to handle
9.3. THE ASSIGNMENT OPERATOR

Figure 9.1: Private data and public interface functions of the class Person, using byte-by-byte assignment

Figure 9.2: Private data and public interface functions of the class Person, using the 'correct' assignment.
the assignment. The purpose of this member function would be to create a copy of an object having its own name, address and phone strings. Such a member function could be:

```cpp
void Person::assign(Person const &other) {
    // delete our own previously used memory
    delete[] d_name;
    delete[] d_address;
    delete[] d_phone;

    // copy the other Person’s data
    d_name = strdupnew(other.d_name);
    d_address = strdupnew(other.d_address);
    d_phone = strdupnew(other.d_phone);
}
```

Using `assign` we could rewrite the offending function `tmpPerson`:

```cpp
void tmpPerson(Person const &person) {
    Person tmp;

    // tmp (having its own memory) holds a copy of person
    tmp.assign(person);

    // now it doesn’t matter that tmp is destroyed..
}
```

This solution is valid, although it only tackles a symptom. It requires the programmer to use a specific member function instead of the assignment operator. The original problem (assignment produces wild pointers) is still not solved. Since it is hard to ‘strictly adhere to a rule’ a way to solve the original problem is of course preferred.

Fortunately a solution exists using operator overloading: the possibility C++ offers to redefine the actions of an operator in a given context. Operator overloading was briefly mentioned earlier, when the operators `<<` and `>>` were redefined to be used with streams (like `cin`, `cout` and `cerr`), see section 3.1.4.

Overloading the assignment operator is probably the most common form of operator overloading in C++. A word of warning is appropriate, though. The fact that C++ allows operator overloading does not mean that this feature should indiscriminately be used. Here’s what you should keep in mind:

- operator overloading should be used in situations where an operator has a defined action, but this default action has undesired side effects in a given context. A clear example is the above assignment operator in the context of the class `Person`.
- operator overloading can be used in situations where the operator is commonly applied and no surprise is introduced when it’s redefined. An example where operator overloading is appropriately used is found in the class `std::string`: assigning one string object to another provides the destination string with a copy of the contents of the source string. No surprises here.
- in all other cases a member function should be defined instead of redefining an operator.

An operator should simply do what it is designed to do. The phrase that’s often encountered in the context of operator overloading is *do as the ints do*. The way operators behave when applied to ints
9.3. THE ASSIGNMENT OPERATOR

is what is expected, all other implementations probably cause surprises and confusion. Therefore, overloading the insertion (<<) and extraction (>>) operators in the context of streams is probably ill-chosen: the stream operations have nothing in common with bitwise shift operations.

9.3.1.1 The member 'operator='

To add operator overloading to a class, the class interface is simply provided with a (usually public) member function naming the particular operator. That member function is thereupon implemented.

To overload the assignment operator =, a member operator=(Class const &rhs) is added to the class interface. Note that the function name consists of two parts: the keyword operator, followed by the operator itself. When we augment a class interface with a member function operator=, then that operator is redefined for the class, which prevents the default operator from being used. In the previous section the function assign was provided to solve the problems resulting from using the default assignment operator. Rather than using an ordinary member function C++ commonly uses a dedicated operator generalizing the operator's default behavior to the class in which it is defined.

The assign member mentioned before may be redefined as follows (the member operator= presented below is a first, rather unsophisticated, version of the overloaded assignment operator. It will shortly be improved):

```cpp
class Person
{
  public: // extension of the class Person
    // earlier members are assumed.
    void operator=(Person const &other);
};
```

Its implementation could be

```cpp
void Person::operator=(Person const &other)
{
  delete[] d_name; // delete old data
  delete[] d_address;
  delete[] d_phone;

  d_name = strdupnew(other.d_name); // duplicate other's data
  d_address = strdupnew(other.d_address);
  d_phone = strdupnew(other.d_phone);
}
```

This member’s actions are similar to those of the previously mentioned member assign, but this member is automatically called when the assignment operator = is used. Actually there are two ways to call overloaded operators as shown in the next example:

```cpp
void tmpPerson(Person const &person)
{
  Person tmp;

  tmp = person;
  tmp.operator=(person); // the same thing
}
```
Overloaded operators are seldom called explicitly, but explicit calls must be used (rather than using the plain operator syntax) when you explicitly want to call the overloaded operator from a pointer to an object (it is also possible to dereference the pointer first and then use the plain operator syntax, see the next example):

```cpp
void tmpPerson(Person const &person)
{
    Person *tmp = new Person;
    tmp->operator=(person);
    *tmp = person;  // yes, also possible...
    delete tmp;
}
```

### 9.4 The ‘this’ pointer

A member function of a given class is always called in combination with an object of its class. There is always an implicit ‘substrate’ for the function to act on. C++ defines a keyword, `this`, to reach this substrate.

The `this` keyword is a pointer variable that always contains the address of the object for which the member function was called. The `this` pointer is implicitly declared by each member function (whether public, protected, or private). The `this` pointer is a constant pointer to an object of the member function’s class. For example, the members of the class `Person` implicitly declare:

```cpp
extern Person *const this;
```

A member function like `Person::name` could be implemented in two ways: with or without using the `this` pointer:

```cpp
char const *Person::name() const  // implicitly using ‘this’
{
    return d_name;
}
```

```cpp
char const *Person::name() const  // explicitly using ‘this’
{
    return this->d_name;
}
```

The `this` pointer is seldom explicitly used, but situations do exist where the `this` pointer is actually required (cf. chapter 16).

### 9.4.1 Sequential assignments and this

C++’s syntax allows for sequential assignments, with the assignment operator associating from right to left. In statements like:

```cpp
a = b = c;
```
the expression \( b = c \) is evaluated first, and its result in turn is assigned to \( a \).

The implementation of the overloaded assignment operator we've encountered thus far does not permit such constructions, as it returns \( \text{void} \).

This imperfection can easily be remedied using the \textit{this} pointer. The overloaded assignment operator expects a reference to an object of its class. It can also return a reference to an object of its class. This reference can then be used as an argument in sequential assignments.

The overloaded assignment operator commonly returns a reference to the current object (i.e., \( \ast \text{this} \)). The next version of the overloaded assignment operator for the class \texttt{Person} thus becomes:

```cpp
Person &Person::operator=(Person const &other) {
    delete[] d_address;
    delete[] d_name;
    delete[] d_phone;

    d_address = strdupnew(other.d_address);
    d_name = strdupnew(other.d_name);
    d_phone = strdupnew(other.d_phone);

    // return current object as a reference
    return *this;
}
```

Overloaded operators may themselves be overloaded. Consider the \texttt{string} class, having overloaded assignment operators \( \text{operator=}=(\text{std::string const &rhs}) \), \( \text{operator=}=(\text{char const *rhs}) \), and several more overloaded versions. These additional overloaded versions are there to handle different situations which are, as usual, recognized by their argument types. These overloaded versions all follow the same mold: when necessary dynamically allocated memory controlled by the object is deleted; new values are assigned using the overloaded operator's parameter values and \( \ast \text{this} \) is returned.

### 9.5 The copy constructor: initialization vs. assignment

Consider the class \texttt{Strings}, introduced in section \texttt{9.2}, once again. As it contains several primitive type data members as well as a pointer to dynamically allocated memory it needs a constructor, a destructor, and an overloaded assignment operator. In fact the class offers two constructors: in addition to the default constructor it offers a constructor expecting a \texttt{char const *const *} and a \texttt{size_t}.

Now consider the following code fragment. The statement references are discussed following the example:

```cpp
int main(int argc, char **argv) {
    Strings s1(argv, argc);  // (1)
    Strings s2;              // (2)
    Strings s3(s1);          // (3)
    s2 = s1;                 // (4)
}
```
• At 1 we see an initialization. The object `s1` is initialized using `main`'s parameters: `Strings`'s second constructor is used.

• At 2 `Strings`'s default constructor is used, initializing an empty `Strings` object.

• At 3 yet another `Strings` object is created, using a constructor accepting an existing `Strings` object. This form of initializations has not yet been discussed. It is called a copy construction and the constructor performing the initialization is called the copy constructor. Copy constructions are also encountered in the following form:

```
Strings s3 = s1;
```

This is a construction and therefore an initialization. It is not an assignment as an assignment needs a left-hand operand that has already been defined. C++ allows the assignment syntax to be used for constructors having only one parameter. It is somewhat deprecated, though.

• At 4 we see a plain assignment.

In the above example three objects were defined, each using a different constructor. The actually used constructor was deduced from the constructor's argument list.

The copy constructor encountered here is new. It does not result in a compilation error even though it hasn’t been declared in the class interface. This takes us to the following rule:

A copy constructor is (almost) always available, even if it isn’t declared in the class’s interface.

The reason for the ‘(almost)’ is given in section 9.7.1.

The copy constructor made available by the compiler is also called the trivial copy constructor. Starting with the C++11 standard it can easily be suppressed (using the `= delete` idiom). The trivial copy constructor performs a byte-wise copy operation of the existing object’s primitive data to the newly created object, calls copy constructors to intialize the object’s class data members from their counterparts in the existing object and, when inheritance is used, calls the copy constructors of the base class(es) to initialize the new object’s base classes.

Consequently, in the above example the trivial copy constructor is used. As it performs a byte-by-byte copy operation of the object’s primitive type data members that is exactly what happens at statement 3. By the time `s3` ceases to exist its destructor deletes its array of strings. Unfortunately `d_string` is of a primitive data type and so it also deletes `s1`'s data. Once again we encounter wild pointers as a result of an object going out of scope.

The remedy is easy: instead of using the trivial copy constructor a copy constructor must explicitly be added to the class’s interface and its definition must prevent the wild pointers, comparably to the way this was realized in the overloaded assignment operator. An object’s dynamically allocated memory is duplicated, so that it contains its own allocated data. The copy constructor is simpler than the overloaded assignment operator in that it doesn’t have to delete previously allocated memory. Since the object is going to be created no previously allocated memory already exists.

Strings's copy constructor can be implemented as follows:

```
Strings::Strings(Strings const &other) :
    d_string(new string[other.d_size]),
    d_size(other.d_size)
{
```
for (size_t idx = 0; idx != d_size; ++idx)
    d_string[idx] = other.d_string[idx];
}

The copy constructor is always called when an object is initialized using another object of its class. Apart from the plain copy construction that we encountered thus far, here are other situations where the copy constructor is used:

- it is used when a function defines a class type value parameter rather than a pointer or a reference. The function's argument initializes the function's parameter using the copy constructor. Example:

  void process(Strings store) // no pointer, no reference
  {
    store.at(3) = "modified"; // doesn’t modify ‘outer’
  }

  int main(int argc, char**argv)
  {
    Strings outer(argv, argc);
    process(outer);
  }

- it is used when a function defines a class type value return type. Example:

  Strings copy(Strings const &store)
  {
    return store;
  }

Here store is used to initialize copy’s return value. The returned Strings object is a temporary, anonymous object that may be immediately used by code calling copy but no assumptions can be made about its lifetime thereafter.

## 9.6 Revising the assignment operator

The overloaded assignment operator has characteristics also encountered with the copy constructor and the destructor:

- The copying of (private) data occurs (1) in the copy constructor and (2) in the overloaded assignment function.
- Allocated memory is deleted (1) in the overloaded assignment function and (2) in the destructor.

The copy constructor and the destructor clearly are required. If the overloaded assignment operator also needs to return allocated memory and to assign new values to its data members couldn’t the destructor and copy constructor be used for that?

As we’ve seen in our discussion of the destructor (section 9.2) the destructor can explicitly be called, but that doesn’t hold true for the (copy) constructor. But let’s briefly summarize what an overloaded assignment operator is supposed to do:

- It should delete the dynamically allocated memory controlled by the current object;
• It should reassign the current object’s data members using a provided existing object of its class.

The second part surely looks a lot like copy construction. Copy construction becomes even more attractive after realizing that the copy constructor also initializes any reference data members the class might have. Realizing the copy construction part is easy: just define a local object and initialize it using the assignment operator’s const reference parameter, like this:

```cpp
Strings &operator=(Strings const &other)
{
    Strings tmp(other);
    // more to follow
    return *this;
}
```

You may think the optimization `operator=(String tmp)` is attractive, but let’s postpone that for a little while (at least until section 9.7).

Now that we’ve done the copying part, what about the deleting part? And isn’t there another slight problem as well? After all we copied all right, but not into our intended (current, `*this`) object.

At this point it’s time to introduce swapping. Swapping two variables means that the two variables exchange their values. We’ll discuss swapping in detail in the next section, but let’s for now assume that we’ve added a member `swap(Strings &other)` to our class Strings. This allows us to complete String’s `operator=` implementation:

```cpp
Strings &operator=(Strings const &other)
{
    Strings tmp(other);
    swap(tmp);
    return *this;
}
```

This implementation of `operator=` is generic: it can be applied to every class whose objects are swappable. How does it work?

• The information in the `other` object is used to initialize a local `tmp` object. This takes care of the copying part of the assignment operator;
• Calling `swap` ensures that the current object receives its new values (with `tmp` receiving the current object’s original values);
• When `operator=` terminates its local `tmp` object ceases to exist and its destructor is called. As it by now contains the data previously owned by the current object, the current object’s original data are now destroyed, effectively completing the destruction part of the assignment operation.

Nice?

### 9.6.1 Swapping

Many classes (e.g., `std::string`) offer `swap` members allowing us to swap two of their objects. The Standard Template Library (STL, cf. chapter 18) offers various functions related to swapping.
There is even a *swap* generic algorithm (cf. section 19.1.61), which is commonly implemented using the assignment operator. When implementing a *swap* member for our class *Strings* it could be used, provided that all of *String*’s data members can be swapped. As this is true (*why* this is true is discussed shortly) we can augment class *Strings* with a *swap* member:

```cpp
void Strings::swap(Strings &other)
{
    swap(d_string, other.d_string);
    swap(d_size, other.d_size);
}
```

Having added this member to *Strings* the copy-and-swap implementation of *String*::operator= can now be used.

When two variables (e.g., `double one` and `double two`) are swapped, each one holds the other one’s value after the swap. So, if `one == 12.50` and `two == -3.14` then after `swap(one, two)` `one == -3.14` and `two == 12.50`.

Variables of primitive data types (pointers and the built-in types) can be swapped, class-type objects can be swapped if their classes offer a *swap* member.

So should we provide our classes with a *swap* member, and if so, how should it be implemented?

The above example (*Strings::swap*) shows the standard way to implement a *swap* member: each of its data members are swapped in turn. But there are situations where a class cannot implement a *swap* member this way, even if the class only defines data members of primitive data types. Consider the situation depicted in figure 9.3.

In this figure there are four objects, each object has a pointer pointing to the next object. The basic organization of such a class looks like this:

```cpp
class List
{
    List *d_next;
    ...
};
```

Initially four objects have their `d_next` pointer set to the next object: 1 to 2, 2 to 3, 3 to 4. This is shown in the upper half of the figure. At the bottom half it is shown what happens if objects 2 and 3...
are swapped: 3’s \( d_{\text{next}} \) point is now at object 2, which still points to 4; 2’s \( d_{\text{next}} \) pointer points to 3’s address, but 2’s \( d_{\text{next}} \) is now at object 3, which is therefore pointing to itself. Bad news!

Another situation where swapping of objects goes wrong happens with classes having data members pointing or referring to data members of the same object. Such a situation is shown in figure 9.4.

Here, objects have two data members, as in the following class setup:

```cpp
class SelfRef
{
    size_t *d_ownPtr; // initialized to &d_data
    size_t d_data;
};
```

The top-half of figure 9.4 shows two objects; their upper data members pointing to their lower data members. But if these objects are swapped then the situation shown in the figure’s bottom half is encountered. Here the values at addresses a and c are swapped, and so, rather than pointing to their bottom data members they suddenly point to other object’s data members. Again: bad news.

The common cause of these failing swapping operations is easily recognized: simple swapping operations must be avoided when data members point or refer to data that is involved in the swapping. If, in figure 9.4 the a and c data members would point to information outside of the two objects (e.g., if they would point to dynamically allocated memory) then the simple swapping would succeed.

However, the difficulty encountered with swapping SelfRef objects does not imply that two SelfRef objects cannot be swapped; it only means that we must be careful when designing swap members. Here is an implementation of SelfRef::swap:

```cpp
void SelfRef::swap(SelfRef &other)
{
    swap(d_data, other.d_data);
}
```
9.6. REVISING THE ASSIGNMENT OPERATOR

In this implementation swapping leaves the self-referential data member as-is, and merely swaps the remaining data. A similar swap member could be designed for the linked list shown in figure 9.3.

9.6.1.1 Fast swapping

As we’ve seen with placement new objects can be constructed in blocks of memory of sizeof(Class) bytes large. And so, two objects of the same class each occupy sizeof(Class) bytes.

If objects of our class can be swapped, and if our class’s data members do not refer to data actually involved in the swapping operation then a very fast swapping method that is based on the fact that we know how large our objects are can be implemented.

In this fast-swap method we merely swap the contents of the sizeof(Class) bytes. This procedure may be applied to classes whose objects may be swapped using a member-by-member swapping operation and can (in practice, although this probably overextends the allowed operations as described by the C++ ANSI/ISO standard) also be used in classes having reference data members.

It simply defines a buffer of sizeof(Class) bytes and performs a circular memcpy operation. Here is its implementation for a hypothetical class Class. It results in very fast swapping:

```cpp
#include <cstring>

void Class::swap(Class &other) {
    char buffer[sizeof(Class)];
    memcpy(buffer, &other, sizeof(Class));
    memcpy(&other, this, sizeof(Class));
    memcpy(this, buffer, sizeof(Class));
}
```

Here is a simple example of a class defining a reference data member and offering a swap member implemented like the one above. The reference data members are initialized to external streams. After running the program one contains two hellos to 1 lines, two contains two hellos to 2 lines (for brevity all members of Reference are defined inline):

```cpp
#include <fstream>
#include <cstring>

class Reference
{
    std::ostream &d_out;

public:
    Reference(std::ostream &out) :
        d_out(out) {}
    void swap(Reference &other) {
        char buffer[sizeof(Reference)];
        memcpy(buffer, this, sizeof(Reference));
        memcpy(this, &other, sizeof(Reference));
        memcpy(&other, buffer, sizeof(Reference));
    }
};
```
std::ostream &out()  
{  
    return d_out;  
}  
};

int main()  
{  
    std::ofstream one("one");  
    std::ofstream two("two");  

    Reference ref1(one);  
    Reference ref2(two);  

    ref1.out() << "hello to 1\n";  
    ref2.out() << "hello to 2\n";  

    ref1.swap(ref2);  

    ref2.out() << "hello to 1\n";  
    ref1.out() << "hello to 2\n";  
}  

Fast swapping should only be used for self-defined classes for which it can be proven that fast-swapping does not corrupt its objects, when swapped.

9.7  Moving data

Before the advent of the C++11 standard C++ offered basically two ways to assign the information pointed to by a data member of a temporary object to an lvalue object. Either a copy constructor or reference counting had to be used. The C++11 standard adds move semantics to these two, allowing transfer of the data pointed to by a temporary object to its destination.

Moving information is based on the concept of anonymous (temporary) data. Temporary values are returned by functions like operator-() and operator+(Type const &lhs, Type const &rhs), and in general by functions returning their results 'by value' instead of returning references or pointers.

Anonymous values are always short-lived. When the returned values are primitive types (int, double, etc.) nothing special happens, but if a class-type object is returned by value then its destructor can be called immediately following the function call that produced the value. In any case, the value itself becomes inaccessible immediately after the call. Of course, a temporary return value may be bound to a reference (lvalue or rvalue), but as far as the compiler is concerned the value now has a name, which by itself ends its status as a temporary value.

In this section we concentrate on anonymous temporary values and show how they can be used to improve the efficiency of object construction and assignment. These special construction and assignment methods are known as move construction and move assignment. Classes supporting move operations are called move-aware.

Classes allocating their own memory usually benefit from becoming move-aware. But a class does not have to use dynamic memory allocation before it can benefit from move operations. Most classes
using composition (or inheritance where the base class uses composition) can benefit from move operations as well.

Movable parameters for class `Class` take the form `Class &&tmp`. The parameter is a `rvalue reference`, and a rvalue reference only binds to an anonymous temporary value. The compiler is required to call functions offering movable parameters whenever possible. This happens when the class defines functions supporting `Class &&` parameters and an anonymous temporary value is passed to such functions. Once a temporary value has a name (which already happens inside functions defining `Class const &` or `Class &&tmp` parameters as within such functions the names of these parameters are available) it is no longer an anonymous temporary value, and within such functions the compiler no longer calls functions expecting anonymous temporary values when the parameters are used as arguments.

The next example (using inline member implementations for brevity) illustrates what happens if a non-const object, a temporary object and a const object are passed to functions `fun` for which these kinds of parameters were defined. Each of these functions call a function `gun` for which these kinds of parameters were also defined. The first time `fun` is called it (as expected) calls `gun(Class &)` Then `fun(Class &&)` is called as its argument is an anonymous (temporary) object. However, inside `fun` the anonymous value has received a name, and so it isn’t anonymous anymore. Consequently, `gun(Class &)` is called once again. Finally `fun(Class const &)` is called, and (as expected) `gun(Class const &)` is now called.

```cpp
#include <iostream>

using namespace std;

class Class
{
    public:
        Class()
        {};

    void fun(Class const &other)
    {
        cout << "fun: Class const &\n";
        gun(other);
    }

    void fun(Class &other)
    {
        cout << "fun: Class &\n";
        gun(other);
    }

    void fun(Class &&tmp)
    {
        cout << "fun: Class &&\n";
        gun(tmp);
    }

    void gun(Class const &other)
    {
        cout << "gun: Class const &\n";
    }

    void gun(Class &other)
    {
        cout << "gun: Class &\n";
    }

    void gun(Class &&tmp)
    {
        cout << "gun: Class &&\n";
    }
};
```
Generally it is pointless to define a function having an rvalue reference return type. The compiler decides whether or not to use an overloaded member expecting an rvalue reference on the basis of the provided argument. If it is an anonymous temporary it calls the function defining the rvalue reference parameter, if such a function is available. A rvalue reference return type is used, e.g., with the \texttt{std::move} call, to keep the rvalue reference nature of its argument, which is known to be a temporary anonymous object. Such a situation can be exploited also in a situation where a temporary object is passed to (and returned from) a function which must be able to modify the temporary object. The alternative, passing a \texttt{const &}, is less attractive as it requires a \texttt{const_cast} before the object can be modified. Here is an example:

\begin{verbatim}
std::string &&doubleString(std::string &&tmp)
{
  tmp += tmp;
  return std::move(tmp);
}
\end{verbatim}

This allows us to do something like

\begin{verbatim}
std::cout << doubleString(std::string("hello "));
\end{verbatim}

to insert \texttt{hello hello} into \texttt{cout}.

The compiler, when selecting a function to call applies a fairly simple algorithm, and also considers copy elision. This is covered shortly (section 9.8).

### 9.7.1 The move constructor (dynamic data)

Our class \texttt{Strings} has, among other members a data member \texttt{string *d_string}. Clearly, \texttt{Strings} should define a copy constructor, a destructor and an overloaded assignment operator.

Now consider the following function \texttt{loadStrings(std::istream &in)} extracting the strings for a \texttt{Strings} object from \texttt{in}. Next, the \texttt{Strings} object filled by \texttt{loadStrings} is returned by value. The function \texttt{loadStrings} returns a temporary object, which can then used to initialize an external \texttt{Strings} object:

\begin{verbatim}
Strings loadStrings(std::istream &in)
\end{verbatim}
9.7. MOVING DATA

```cpp
{    
    Strings ret;    
    // load the strings into 'ret'
    return ret;  
}  

// usage:
Strings store(loadStrings(cin));
```

In this example two full copies of a `Strings` object are required:

- initializing `loadString`'s value return type from its local `Strings` `ret` object;
- initializing `store` from `loadString`'s return value

We can improve the above procedure by defining a move constructor. Here is the declaration of the `Strings` class move constructor:

```cpp
Strings(Strings &&tmp);
```

Move constructors of classes using dynamic memory allocation are allowed to assign the values of pointer data members to their own pointer data members without requiring them to make a copy of the source's data. Next, the temporary's pointer value is set to zero to prevent its destructor from destroying data now owned by the just constructed object. The move constructor has `grabbed or stolen` the data from the temporary object. This is OK as the temporary object cannot be referred to again (as it is anonymous, it cannot be accessed by other code) and the temporary objects cease to exist shortly after the constructor's call. Here is the implementation of `Strings` move constructor:

```cpp
Strings::Strings(Strings &&tmp) :
    d_memory(tmp.d_memory),
    d_size(tmp.d_size),
    d_capacity(tmp.d_capacity) {
    tmp.d_memory = 0;
}
```

In section 9.5 it was stated that the copy constructor is almost always available. *Almost* always as the declaration of a move constructor suppresses the default availability of the copy constructor. The default copy constructor is also suppressed if a move assignment operator is declared (cf. section 9.7.3).

The following example shows a simple class `Class`, declaring a move constructor. In the main function following the class interface a `Class` object is defined which is then passed to the constructor of a second `Class` object. Compilation fails with the compiler reporting:

```
error: cannot bind 'Class' lvalue to 'Class&&'
error: initializing argument 1 of 'Class::Class(Class&&)'
```

```cpp
class Class
{
    public:
        Class() = default;
```


```cpp
Class(Class &&tmp)
{
};

int main()
{
    Class one;
    Class two(one);
}
```

The cure is easy: after declaring a (possibly default) copy constructor the error disappears:

```cpp
class Class
{
    public:
        Class() = default;
        Class(Class const &other) = default;
        Class(Class &&tmp)
        {}
};

int main()
{
    Class one;
    Class two(one);
}
```

### 9.7.2 The move constructor (composition)

Classes not using pointer members pointing to memory controlled by its objects (and not having base classes doing so, see chapter 13) may also benefit from overloaded members expecting rvalue references. The class benefits from move operations when one or more of the composed data members themselves support move operations.

Move operations cannot be implemented if the class type of a composed data member does not support moving or copying. Currently, stream classes fall into this category.

An example of a move-aware class is the class `std::string`. A class `Person` could use composition by defining `std::string d_name` and `std::string d_address`. Its move constructor would then have the following prototype:

```cpp
Person(Person &&tmp);
```

However, the following implementation of this move constructor is incorrect:

```cpp
Person::Person(Person &&tmp)
:
    d_name(tmp.d_name),
    d_address(tmp.d_address)
{};
```

It is incorrect as it `string`'s copy constructors rather than `string`'s move constructors are called. If you’re wondering why this happens then remember that move operations are only performed for
9.7. MOVING DATA

anonymous objects. To the compiler anything having a name isn’t anonymous. And so, by implication, having available a rvalue reference does not mean that we’re referring to an anonymous object. But we know that the move constructor is only called for anonymous arguments. To use the corresponding string move operations we have to inform the compiler that we’re talking about anonymous data members as well. For this a cast could be used (e.g., `static_cast<Person &&>(tmp)`), but the C++-0x standard provides the function `std::move` to anonymize a named object. The correct implementation of `Person`’s move construction is, therefore:

```cpp
Person::Person(Person &&tmp)
 :
   d_name( std::move(tmp.d_name) ),
   d_address( std::move(tmp.d_address) )
{}
```

The function `std::move` is (indirectly) declared by many header files. If no header is already declaring `std::move` then include `utility`.

When a class using composition not only contains class type data members but also other types of data (pointers, references, primitive data types), then these other data types can be initialized as usual. Primitive data type members can simply be copied; references can be initialized as usual and pointers may use move operations as discussed in the previous section.

The compiler never calls move operations for variables having names. Let’s consider the implications of this by looking at the next example, assuming `Class` offers a move constructor and a copy constructor:

```cpp
Class factory();

void fun(Class const &other); // a
void fun(Class &&tmp);        // b

void callee(Class &&tmp);
{
   fun(tmp);                   // 1
}

int main()
{
   callee(factory());
}
```

- At 1 function a is called. At first sight this might be surprising, but `fun`’s argument is not an anonymous temporary object but a named temporary object.

Realizing that `fun(tmp)` might be called twice the compiler’s choice is understandable. If `tmp`’s data would have been grabbed at the first call, the second call would receive `tmp` without any data. But at the last call we might know that `tmp` is never used again and so we might like to ensure that `fun(Class &&)` is called. For this, once again, `std::move` is used:

```cpp
fun(std::move(tmp));          // last call!
```
9.7.3 Move-assignment

In addition to the overloaded assignment operator a move assignment operator may be implemented for classes supporting move operations. In this case, if the class supports swapping the implementation is surprisingly simple. No copy construction is required and the move assignment operator can simply be implemented like this:

```cpp
Class &operator=(Class &&tmp)
{
    swap(tmp);
    return *this;
}
```

If swapping is not supported then the assignment can be performed for each of the data members in turn, using `std::move` as shown in the previous section with a class `Person`. Here is an example showing how to do this with that class `Person`:

```cpp
Person &operator=(Person &&tmp)
{
    d_name = std::move(tmp.d_name);
    d_address = std::move(tmp.d_address);
    return *this;
}
```

As noted previously (section 9.7.1) declaring a move assignment operator suppresses the default availability of the copy constructor. It is made available again by declaring the copy constructor in the class's interface (and of course by providing an explicit implementation or by using the `= default` default implementation).

9.7.4 Revising the assignment operator (part II)

Now that we've familiarized ourselves with the overloaded assignment operator and the move-assignment, let's once again have a look at their implementations for a class `Class`, supporting swapping through its `swap` member. Here is the generic implementation of the overloaded assignment operator:

```cpp
Class &operator=(Class const &other)
{
    Class tmp(other);
    swap(tmp);
    return *this;
}
```

and this is the move-assignment operator:

```cpp
Class &operator=(Class &&tmp)
{
    swap(tmp);
    return *this;
}
```
They look remarkably similar in the sense that the overloaded assignment operator's code is identical to the move-assignment operator's code once a copy of the `other` object is available. Since the overloaded assignment operator's `tmp` object really is nothing but a temporary `Class` object we can use this fact by implementing the overloaded assignment operator in terms of the move-assignment. Here is a second revision of the overloaded assignment operator:

```cpp
Class &operator=(Class const &other) {
    Class tmp(other);
    return *this = std::move(tmp);
}
```

### 9.7.5 Moving and the destructor

Once a class becomes a *move-aware* class one should realize that its destructor still performs its job as implemented. Consequently, when moving pointer values from a temporary source to a destination the move constructor should make sure that the temporary’s pointer value is set to zero, to prevent doubly freeing memory.

If a class defines pointers to pointer data members there usually is not only a pointer that is moved, but also a `size_t` defining the number of elements in the array of pointers.

Once again, consider the class `Strings`. Its destructor is implemented like this:

```cpp
Strings::~Strings() {
    for (string **end = d_string + d_size; end-- != d_string; )
        delete *end;
    delete[] d_string;
}
```

The move constructor (and other move operations!) must realize that the destructor not only deletes `d_string`, but also considers `d_size`. A member implementing move operations should therefore not only set `d_string` to zero but also `d_size`. The previously shown move constructor for `Strings` is therefore incorrect. Its improved implementation is:

```cpp
Strings::Strings(Strings &&tmp) :
    d_memory(tmp.d_memory),
    d_size(tmp.d_size),
    d_capacity(tmp.d_capacity)
{
    tmp.d_memory = 0;
    tmp.d_size = 0;
}
```

If operations by the destructor all depend on `d_string` having a non-zero value then variations of the above approach are of course possible. The move operations merely could decide to set `d_memory` to 0, and then test whether `d_memory == 0` in the destructor (and if so, end the destructor’s actions), saving some `d_size` assignments.
9.7.6 Move-only classes

Classes may very well allow move semantics without offering copy semantics. Most stream classes belong to this category. Extending their definition with move semantics greatly enhances their usability. Once move semantics becomes available for such classes, so called *factory functions* (functions returning an object constructed by the function) can easily be implemented. E.g.,

```cpp
// assume char *filename
ifstream inStream(openIstream(filename));
```

For this example to work an *ifstream* constructor must offer a move constructor. This ensures that only one object refers to the *openIstream*.

Once classes offer move semantics their objects can also safely be stored in standard containers. When such containers perform reallocations (e.g., when their sizes are enlarged) they use the object's move constructors rather than their copy constructors. As move-only classes suppress copy semantics containers storing objects of move-only classes implement the correct behavior in that it is impossible to assign such containers to each other.

9.7.7 Default move constructors and assignment operators

As we’ve seen, classes by default offer a copy constructor and assignment operator. These class members are implemented so as to provide basic support: data members of primitive data types are copied byte-by-byte, but for class type data members their corresponding copy constructors c.q. assignment operators are called.

The compiler can provide default implementations for move constructors and move assignment operators.

However, except for the copy constructor, default implementations for constructors (c.q. assignment operators) are no longer provided once a class declares at least one constructor (c.q. assignment operator), while the default copy constructor is suppressed by declarations of either the move constructor or the move assignment operator.

If default implementations should be available in these cases, it’s easy to add them to the class by adding `= default` to the appropriate constructor and assignment operator declarations.

Here is an example of a class offering all defaults: constructor, copy constructor, move constructor, assignment operator and move assignment operator:

```cpp
class Defaults
{
    int d_x;
    Mov d_mov;
};
```

Assuming that *Mov* is a class offering move operations in addition to the standard deep copy operations, then the following actions are performed on the destination's *d_mov* and *d_x*:

```cpp
Defaults factory();

int main()
{
    Mov operation: d_x:
```
9.7. MOVING DATA

Defaults one;
Defaults two(one);
Defaults three(factory());

one = two;

one = factory();

If, however, Defaults declares at least one constructor (it could be the copy constructor) and one assignment operator, then only those members and the copy constructor remain available. E.g.:

```cpp
class Defaults {
  int d_x;
  Mov d_mov;

  public:
    Defaults(int x);
    Defaults(Defaults &&tmp);

  Defaults factory();

  int main()
  {
   Defaults one;  ERROR: not available
   Defaults two(one);  MOV(Mov const &), one.d_x
   Defaults three(factory());  ERROR: not available

   one = two;  ERROR: not available
   one = factory();  MOV::operator=( tmp.d_x
                      Mov &&tmp)
  }
To reestablish the defaults, append = default to the appropriate declarations:

class Defaults {
  int d_x;
  Mov d_mov;

  public:
    Defaults() = default;
    Defaults(Defaults &&tmp) = default;
    Defaults(int x);
    // Default(Default const &) remains available
    Defaults operator=(Defaults const &rhs) = default;
```
Defaults operator=(Defaults &&tmp);
};

Be cautious, declaring defaults, as default implementations copy data members of primitive types byte-by-byte from the source object to the destination object. This is likely to cause problems with pointer type data members.

The = default suffix can only be used when declaring constructors or assignment operators in the class’s public section.

### 9.7.8 Moving: implications for class design

Here are some general rules to apply when designing classes offering value semantics (i.e., classes whose objects can be used to initialize other objects of their class and that can be assigned to other objects of their class):

- Classes using pointers to dynamically allocated memory, owned by the class’s objects must be provided with a copy constructor, an overloaded copy assignment operator and a destructor;
- Classes using pointers to dynamically allocated memory, owned by the class’s objects, should be provided with a move constructor and a move assignment operator;
- Classes using composition may benefit from move constructors and move assignment operators as well. Some classes support neither move nor copy construction and assignment (for example: stream classes don’t). If your class contains data members of such class types then defining move operations is pointless.

In the previous sections we’ve also encountered an important design principle that can be applied to move-aware classes:

> Whenever a member of a class receives a `const &` to an object of its own class and creates a copy of that object to perform its actual actions on, then that function’s implementation can be implemented by an overloaded function expecting an rvalue reference.

The former function can now call the latter by passing `std::move(tmp)` to it. The advantages of this design principle should be clear: there is only one implementation of the actual actions, and the class automatically becomes move-aware with respect to the involved function.

We’ve seen an initial example of the use of this principle in section 9.7.4. Of course, the principle cannot be applied to the copy constructor itself, as you need a copy constructor to make a copy. The copy- and move constructors must always be implemented independently from each other.

### 9.8 Copy Elision and Return Value Optimization

When the compiler selects a member function (or constructor) it applies a simple set of rules, matching arguments with parameter types.

Below two tables are shown. The first table should be used in cases where a function argument has a name, the second table should be used in cases where the argument is anonymous. In each table select the const or non-const column and then use the topmost overloaded function that is available having the specified parameter type.
The tables do not handle functions defining value parameters. If a function has overloads expecting, respectively, a value parameter and some form of reference parameter the compiler reports an ambiguity when such a function is called. In the following selection procedure we may assume, without loss of generality, that this ambiguity does not occur and that all parameter types are reference parameters.

Parameter types matching a function’s argument of type $T$ if the argument is:

- a named argument (an lvalue or a named rvalue)

<table>
<thead>
<tr>
<th>the argument is:</th>
<th>non-const</th>
<th>const</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use the topmost available function</td>
<td>$(T &amp;)$</td>
<td>$(T \text{ const } &amp;)$</td>
</tr>
</tbody>
</table>

Example: for an `int x` argument a function `fun(int \&)` is selected rather than a function `fun(int \text{ const } \&)`.

If no `fun(int \&)` is available the `fun(int \text{ const } \&)` function is used. If neither is available the compiler reports an error.

- an anonymous argument (an anonymous temporary or a literal value)

<table>
<thead>
<tr>
<th>the argument is:</th>
<th>non-const</th>
<th>const</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use the topmost available function</td>
<td>$(T &amp;&amp;)$</td>
<td>$(T \text{ const } &amp;&amp;)$</td>
</tr>
<tr>
<td>available function</td>
<td>$(T \text{ const } &amp;&amp;)$</td>
<td>$(T \text{ const } &amp;&amp;)$</td>
</tr>
</tbody>
</table>

Example: when the return value of an `int arg()` function is passed to a function `fun` for which various overloaded versions are available `fun(int \&\&)` is selected. If this function is unavailable but `fun(int \text{ const } \&\&)` is, then the latter function is used. If none of these two functions is available the compiler reports an error.

The tables show that eventually all arguments can be used with a function specifying a $T$ \text{ const &} parameter. For anonymous arguments a similar catch all is available having a higher priority: $T$ \text{ const &\&} matches all anonymous arguments. Functions having this signature are normally not defined as their implementations are (should be) identical to the implementations of the functions expecting a $T$ \text{ const &} parameter. Since the temporary can apparently not be modified a function defining a $T$ \text{ const &\&} parameter has no alternative but to copy the temporary’s resources. As this task is already performed by functions expecting a $T$ \text{ const &}, there is no need for implementing functions expecting $T$ \text{ const &\&} parameters.

As we’ve seen the move constructor grabs the information from a temporary for its own use. That is OK as the temporary is going to be destroyed after that anyway. It also means that the temporary’s data members are modified.

Having defined appropriate copy and/or move constructors it may be somewhat surprising to learn that the compiler may decide to stay clear of a copy or move operation. After all making no copy and not moving is more efficient than copying or moving.

The option the compiler has to avoid making copies (or perform move operations) is called \textit{copy elision or return value optimization}. In all situations where copy or move constructions are appropriate the compiler may apply copy elision. Here are the rules. In sequence the compiler considers the following options, stopping once an option can be selected:

- if a copy or move constructor exists, try copy elision
if a move constructor exists, move.
• if a copy constructor exists, copy.
• report an error

All modern compilers apply copy elision. Here are some examples where it may be encountered:

class Elide;

Elide fun() // 1
{
    Elide ret;
    return ret;
}

void gun(Elide par);

Elide elide(fun()); // 2

gun(fun()); // 3

• At 1 ret may never exist. Instead of using ret and copying ret eventually to fun's return value it may directly use the area used to contain fun's return value.
• At 2 fun's return value may never exist. Instead of defining an area containing fun's return value and copying that return value to elide the compiler may decide to use elide to create fun's return value in.
• At 3 the compiler may decide to do the same for gun's par parameter: fun's return value is directly created in par's area, thus eliding the copy operation from fun's return value to par.

9.9 Plain Old Data

C++ inherited the struct concept from C and extended it with the class concept. Structs are still used in C++, mainly to store and pass around aggregates of different data types. A commonly term for these structs is plain old data (pod). Plain old data is commonly used in C++ programs to aggregate data. E.g., when a function needs to return a double, bool and std::string these three different data types may be aggregated using a struct that merely exists to pass along values. Data protection and functionality is hardly ever an issue. For such cases C and C++ use structs. But as a C++ struct is just a class with special access rights some members (constructors, destructor, overloaded assignment operator) may implicitly be defined. The plain old data capitalizes on this concept by requiring that its definition remains as simple as possible. Specifically the C++11 standard considers pod to be a class or struct having the following characteristics:

• it has a trivial default constructor.
  If a type has some trivial member then the type (or its base class(es), cf. chapter 13) does not explicitly define that member. Rather, it is supplied by the compiler. A trivial default constructor leaves all its non-class data members uninitialized and calls the default constructors of all its class data members. A class having a trivial default constructor does not define any constructor at all (nor does/do its base class/classes). It may also define the default constructor using the default constructor syntax introduced in section 7.6;
9.10. CONCLUSION

- it has a trivial copy constructor.
  A trivial copy constructor byte-wise copies the non-class data members from the provided existing class object and uses copy constructors to initialize its base class(es) and class data members with the information found in the provided existing class object;

- it has a trivial overloaded assignment operator.
  A trivial assignment operator performs a byte-wise copy of the non-class data members of the provided right-hand class object and uses overloaded assignment operators to assign new values to its class data members using the corresponding members of the provided right-hand class object;

- it has a trivial destructor.
  A trivial destructor calls the destructors of its base class(es) and class-type data members;

- it has a standard layout.

A standard-layout class or struct

- has only non-static data members that are themselves showing the standard-layout;
- has identical access control (public, private, protected) for all its non-static members;

Furthermore, in the context of class derivation (cf. chapters 14 and 13), a standard-layout class or struct:

- has only base classes that themselves show the standard-layout;
- has at most one (in)direct base class having non-static members;
- has no base classes of the same type as its first non-static data member;
- has no virtual base classes;
- has no virtual members.

9.10 Conclusion

Four important extensions to classes were introduced in this chapter: the destructor, the copy constructor, the move constructor and the overloaded assignment operator. In addition the importance of swapping, especially in combination with the overloaded assignment operator, was stressed.

Classes having pointer data members, pointing to dynamically allocated memory controlled by the objects of those classes, are potential sources of memory leaks. The extensions introduced in this chapter implement the standard defense against such memory leaks.

Encapsulation (data hiding) allows us to ensure that the object’s data integrity is maintained. The automatic activation of constructors and destructors greatly enhance our capabilities to ensure the data integrity of objects doing dynamic memory allocation.

A simple conclusion is therefore that classes whose objects allocate memory controlled by themselves must at least implement a destructor, an overloaded assignment operator and a copy constructor. Implementing a move constructor remains optional, but it allows us to use factory functions with classes not allowing copy construction and/or assignment.

In the end, assuming the availability of at least a copy or move constructor, the compiler might avoid them using copy elision. The compiler is free to use copy elision wherever possible; it is,
however, never a requirement. The compiler may therefore always decide not to use copy elision. In all situations where otherwise a copy or move constructor would have been used the compiler may consider to use copy elision.
Chapter 10

Exceptions

C supports several ways for a program to react to situations breaking the normal unhampered flow of a program:

- The function may notice the abnormality and issue a message. This is probably the least disastrous reaction a program may show.

- The function in which the abnormality is observed may decide to stop its intended task, returning an error code to its caller. This is a great example of postponing decisions: now the calling function is faced with a problem. Of course the calling function may act similarly, by passing the error code up to its caller.

- The function may decide that things are going out of hand, and may call exit to terminate the program completely. A tough way to handle a problem if only because the destructors of local objects aren’t activated.

- The function may use a combination of the functions set jmp and long jmp to enforce non-local exits. This mechanism implements a kind of goto jump, allowing the program to continue at an outer level, skipping the intermediate levels which would have to be visited if a series of returns from nested functions would have been used.

In C++ all these flow-breaking methods are still available. However, of the mentioned alternatives, set jmp and long jmp isn’t frequently encountered in C++ (or even in C) programs, due to the fact that the program flow is completely disrupted.

C++ offers exceptions as the preferred alternative to, e.g., set jmp and long jmp. Exceptions allow C++ programs to perform a controlled non-local return, without the disadvantages of long jmp and set jmp.

Exceptions are the proper way to bail out of a situation which cannot be handled easily by a function itself, but which is not disastrous enough for a program to terminate completely. Also, exceptions provide a flexible layer of control between the short-range return and the crude exit.

In this chapter exceptions are covered. First an example is given of the different impact exceptions and the set jmp/long jmp combination have on programs. This example is followed by a discussion of the formal aspects of exceptions. In this part the guarantees our software should be able to offer when confronted with exceptions are presented. Exceptions and their guarantees have consequences for constructors and destructors. We’ll encounter these consequences at the end of this chapter.
10.1 Exception syntax

Before contrasting the traditional C way of handling non-local gotos with exceptions let's introduce the syntactic elements that are involved when using exceptions.

- Exceptions are generated by a `throw` statement. The keyword `throw`, followed by an expression of a certain type, throws the expression value as an exception. In C++ anything having value semantics may be thrown as an exception: an `int`, a `bool`, a `string`, etc. However, there also exists a `standard exception` type (cf. section 10.8) that may be used as base class (cf. chapter 13) when defining new exception types.

- Exceptions are generated within a well-defined local environment, called a `try`-block. The run-time support system ensures that all of the program's code is itself surrounded by a `global try block`. Thus, every exception generated by our code will always reach the boundary of at least one `try`-block. A program terminates when an exception reaches the boundary of the global `try` block, and when this happens destructors of local and global objects that were alive at the point where the exception was generated are not called. This is not a desirable situation and therefore all exceptions should be generated within a `try`-block explicitly defined by the program. Here is an example of a string exception thrown from within a `try`-block:

```cpp
try
{
    // any code can be defined here
    if (someConditionIsTrue)
        throw string("this is the std::string exception");
    // any code can be defined here
}
```

- `catch`: Immediately following the `try`-block, one or more `catch`-clauses must be defined. A `catch`-clause consists of a `catch`-header defining the type of the exception it can catch followed by a compound statement defining what to do with the caught exception:

```cpp
catch (string const &msg)
{
    // statements in which the caught string object are handled
}
```

Multiple `catch` clauses may appear underneath each other, one for each exception type that has to be caught. In general the `catch` clauses may appear in any order, but there are exceptions requiring a specific order. To avoid confusion it's best to put a `catch` clause for the most general exception last. At most one `exception` clause will be activated. C++ does not support a `Java-style finally`-clause activated after completing a catch clause.

10.2 An example using exceptions

In the following examples the same basic program is used. The program uses two classes, `Outer` and `Inner`.

First, an `Outer` object is defined in `main`, and its member `Outer::fun` is called. Then, in `Outer::fun` an `Inner` object is defined. Having defined the `Inner` object, its member `Inner::fun` is called.
That’s about it. The function `Outer::fun` terminates calling `inner`’s destructor. Then the program terminates, activating `outer`’s destructor. Here is the basic program:

```cpp
#include <iostream>
using namespace std;

class Inner {
   public:
      Inner();
      ~Inner();
      void fun();
   };

Inner::Inner()
   { cout << "Inner constructor\n"; }

Inner::~Inner()
   { cout << "Inner destructor\n"; }

void Inner::fun()
   { cout << "Inner fun\n"; }

class Outer {
   public:
      Outer();
      ~Outer();
      void fun();
   };

Outer::Outer()
   { cout << "Outer constructor\n"; }

Outer::~Outer()
   { cout << "Outer destructor\n"; }

void Outer::fun()
   { Inner in;
      cout << "Outer fun\n";
      in.fun();
   }

int main()
   { Outer out;
      out.fun();
   }
```
CHAPTER 10. EXCEPTIONS

Generated output:
Outer constructor
Inner constructor
Outer fun
Inner fun
Inner destructor
Outer destructor
*/

After compiling and running, the program’s output is entirely as expected: the destructors are called in their correct order (reversing the calling sequence of the constructors).

Now let’s focus our attention on two variants in which we simulate a non-fatal disastrous event in the Inner::fun function. This event must supposedly be handled near main’s end.

We’ll consider two variants. In the first variant the event is handled by setjmp and longjmp; in the second variant the event is handled using C++’s exception mechanism.

10.2.1 Anachronisms: ‘setjmp’ and ‘longjmp’

The basic program from the previous section is slightly modified to contain a variable jmp_buf jmpBuf used by setjmp and longjmp.

The function Inner::fun calls longjmp, simulating a disastrous event, to be handled near main’s end. In main a target location for the long jump is defined through the function setjmp. Setjmp’s zero return indicates the initialization of the jmp_buf variable, in which case Outer::fun is called. This situation represents the ‘normal flow’.

The program’s return value is zero only if Outer::fun terminates normally. The program, however, is designed in such a way that this won’t happen: Inner::fun calls longjmp. As a result the execution flow returns to the setjmp function. In this case it does not return a zero return value. Consequently, after calling Inner::fun from Outer::fun main’s if-statement is entered and the program terminates with return value 1. Try to follow these steps when studying the following program source, which is a direct modification of the basic program given in section 10.2:

```cpp
#include <iostream>
#include <setjmp.h>
#include <cstdlib>

using namespace std;

jmp_buf jmpBuf;

class Inner
{
  public:
    Inner();
    ~Inner();
    void fun();
};

Inner::Inner()
{  
```
10.2. AN EXAMPLE USING EXCEPTIONS

```c++
    cout << "Inner constructor\n";
}
void Inner::fun()
{
    cout << "Inner fun\n";
    longjmp(jmpBuf, 0);
}
Inner::~Inner()
{
    cout << "Inner destructor\n";
}

class Outer
{
    public:
        Outer();
        ~Outer();
        void fun();
};
Outer::Outer()
{
    cout << "Outer constructor\n";
}
Outer::~Outer()
{
    cout << "Outer destructor\n";
}
void Outer::fun()
{
    Inner in;
    cout << "Outer fun\n";
    in.fun();
}

int main()
{
    Outer out;
    if (setjmp(jmpBuf) != 0)
        return 1;
    out.fun();
}
/*
   Generated output:
Outer constructor
Inner constructor
Outer fun
Inner fun
Outer destructor
*/
```

This program’s output clearly shows that `Inner`’s destructor is not called. This is a direct conse-
sequence of the non-local jump performed by `longjmp`. Processing proceeds immediately from the `longjmp` call inside `Inner::fun` to `setjmp` in `main`. There, its return value is unequal zero, and the program terminates with return value 1. Because of the non-local jump `Inner::~Inner` is never executed: upon return to `main`'s `setjmp` the existing stack is simply broken down disregarding any destructors waiting to be called.

This example illustrates that the destructors of objects can easily be skipped when `longjmp` and `setjmp` are used and C++ programs should therefore avoid those functions like the plague.

### 10.2.2 Exceptions: the preferred alternative

Exceptions are C++’s answer to the problems caused by `setjmp` and `longjmp`. Here is an example using exceptions. The program is once again derived from the basic program of section 10.2:

```cpp
#include <iostream>
using namespace std;

class Inner
{
    public:
        Inner();
        ~Inner();
        void fun();
};
Inner::Inner()
{
    cout << "Inner constructor\n";
}
Inner::~Inner()
{
    cout << "Inner destructor\n";
}
void Inner::fun()
{
    cout << "Inner fun\n";
    throw 1;
    cout << "This statement is not executed\n";
}

class Outer
{
    public:
        Outer();
        ~Outer();
        void fun();
};
Outer::Outer()
{
    cout << "Outer constructor\n";
}
Outer::~Outer()
{

10.3. THROWING EXCEPTIONS

```cpp
int main()
{
    Outer out;
    try
    {
        out.fun();
    }
    catch (int x)
    {}
}
/*
 * Generated output:
 * Outer constructor
 * Inner constructor
 * Outer fun
 * Inner fun
 * Inner destructor
 * Outer destructor
 */
```

Inner::fun now throws an int exception where a longjmp was previously used. Since in.fun is called by out.fun, the exception is generated within the try block surrounding the out.fun call. As an int value was thrown this value reappears in the catch clause beyond the try block.

Now Inner::fun terminates by throwing an exception instead of calling longjmp. The exception is caught in main, and the program terminates. Now we see that inner's destructor is properly called. It is interesting to note that Inner::fun's execution really terminates at the throw statement: The cout statement, placed just beyond the throw statement, isn't executed.

What did this example teach us?

- Exceptions provide a means to break a function's (and program's) normal flow without having to use a cascade of return-statements, and without the need to terminate the program using blunt tools like the function exit.

- Exceptions do not disrupt the proper activation of destructors. Since setjmp and longjmp do disrupt the proper activation of destructors their use is strongly deprecated in C++.

10.3 Throwing exceptions

Exceptions are generated by throw statements. The throw keyword is followed by an expression, defining the thrown exception value. Example:

```cpp
    throw "Hello world";   // throws a char *
```
throw 18;   // throws an int
throw string("hello"); // throws a string

Local objects cease to exist when a function terminates. This is no different for exceptions.

Objects defined locally in functions are automatically destroyed once exceptions thrown by these functions leave these functions. This also happens to objects thrown as exceptions. However, just before leaving the function context the object is copied and it is this copy that eventually reaches the appropriate catch clause.

The following examples illustrates this process. Object::fun defines a local Object toThrow, that is thrown as an exception. The exception is caught in main. But by then the object originally thrown doesn't exist anymore, and main received a copy:

```cpp
#include <iostream>
#include <string>
using namespace std;

class Object
{
    string d_name;

public:
    Object(string name)
    :
        d_name(name)
    {
        cout << "Constructor of " << d_name << "\n";
    }
    Object(Object const &other)
    :
        d_name(other.d_name + " (copy)"")
    {
        cout << "Copy constructor for " << d_name << "\n";
    }
    ~Object() {
        cout << "Destructor of " << d_name << "\n";
    }
    void fun() {
        Object toThrow("local object");
        cout << "Calling fun of " << d_name << "\n";
        throw toThrow;
    }
    void hello() {
        cout << "Hello by " << d_name << "\n";
    }
};

int main()
{
    Object out("main object");
    try
```
10.3. THROWING EXCEPTIONS

```cpp
{  
    out.fun();
}
catch (Object o)
{
    cout << "Caught exception\n";
    o.hello();
}
```

Object’s copy constructor is special in that it defines its name as the other object’s name to which the string " (copy)" is appended. This allow us to monitor the construction and destruction of objects more closely. Object::fun generates an exception, and throws its locally defined object. Just before throwing the exception the program has produced the following output:

- Constructor of 'main object'
- Constructor of 'local object'
- Calling fun of 'main object'

When the exception is generated the next line of output is produced:

```
Copy constructor for 'local object' (copy)
```

The local object is passed to throw where it is treated as a value argument, creating a copy of toThrow. This copy is thrown as the exception, and the local toThrow object ceases to exist. The thrown exception is now caught by the catch clause, defining an Object value parameter. Since this is a value parameter yet another copy is created. Thus, the program writes the following text:

- Destructor of 'local object'
- Copy constructor for 'local object' (copy) (copy)

The catch block now displays:

```
Caught exception
```

Following this o’s hello member is called, showing us that we indeed received a copy of the copy of the original toThrow object:

```
Hello by 'local object' (copy) (copy)
```

Then the program terminates and its remaining objects are now destroyed, reversing their order of creation:

- Destructor of 'local object' (copy) (copy)
- Destructor of 'local object' (copy)
- Destructor of 'main object'

The copy created by the catch clause clearly is superfluous. It can be avoided by defining object reference parameters in catch clauses: ‘catch (Object &o)’. The program now produces the following output:

```
Constructor of 'main object'
```
CHAPTER 10. EXCEPTIONS

Constructor of 'local object'
Calling fun of 'main object'
Copy constructor for 'local object' (copy)
Destructor of 'local object'
Caught exception
Hello by 'local object' (copy)
Destructor of 'local object' (copy)
Destructor of 'main object'

Only a single copy of `throw` was created.

It's a bad idea to throw a `pointer` to a locally defined object. The pointer is thrown, but the object to which the pointer refers ceases to exist once the exception is thrown. The catcher receives a wild pointer. Bad news....

Let's summarize the above findings:

- Local objects are thrown as copied objects;

- Don't throw pointers to local objects;

- It is possible to throw pointers to dynamically generated objects. In this case one must take care that the generated object is properly deleted by the exception handler to prevent a memory leak.

Exceptions are thrown in situations where a function can’t complete its assigned task, but the program is still able to continue. Imagine a program offering an interactive calculator. The program expects numeric expressions, which are evaluated. Expressions may show syntactic errors or it may be mathematically impossible to evaluate them. Maybe the calculator allows us to define and use variables and the user might refer to non-existing variables: plenty of reasons for the expression evaluation to fail, and so many reasons for exceptions to be thrown. None of those should terminate the program. Instead, the program’s user is informed about the nature of the problem and is invited to enter another expression. Example:

```cpp
if (!parse(expressionBuffer)) // parsing failed
    throw "Syntax error in expression";

if (!lookup(variableName)) // variable not found
    throw "Variable not defined";

if (divisionByZero()) // unable to do division
    throw "Division by zero is not defined";
```

Where these `throw` statements are located is irrelevant: they may be found deeply nested inside the program, or at a more superficial level. Furthermore, `functions` may be used to generate the exception to be thrown. An `Exception` object might support stream-like insertion operations allowing us to do, e.g.,

```cpp
if (!lookup(variableName))
    throw Exception() << "Undefined variable ‘" << variableName << ";";
```
10.3. THROWING EXCEPTIONS

10.3.1 The empty ‘throw’ statement

Sometimes it is required to inspect a thrown exception. An exception catcher may decide to ignore the exception, to process the exception, to rethrow it after inspection or to change it into another kind of exception. For example, in a server-client application the client may submit requests to the server by entering them into a queue. Normally every request is eventually answered by the server. The server may reply that the request was successfully processed, or that some sort of error has occurred. On the other hand, the server may have died, and the client should be able to discover this calamity, by not waiting indefinitely for the server to reply.

In this situation an intermediate exception handler is called for. A thrown exception is first inspected at the middle level. If possible it is processed there. If it is not possible to process the exception at the middle level, it is passed on, unaltered, to a more superficial level, where the really tough exceptions are handled.

By placing an empty throw statement in the exception handler’s code the received exception is passed on to the next level that might be able to process that particular type of exception. The rethrown exception is never handled by one of its neighboring exception handlers; it is always transferred to an exception handler at a more superficial level.

In our server-client situation a function

\begin{verbatim}
initialExceptionHandler(string &exception)
{
    if (!plainMessage(exception))
        throw;
    handleTheMessage(exception);
}
\end{verbatim}

could be designed to handle the string exception. The received message is inspected. If it’s a simple message it’s processed, otherwise the exception is passed on to an outer level. In initialExceptionHandler’s implementation the empty throw statement is used:

Below (section 10.5), the empty throw statement is used to pass on the exception received by a catch-block. Therefore, a function like initialExceptionHandler can be used for a variety of thrown exceptions, as long as their types match initialExceptionHandler’s parameter, which is a string.

The next example jumps slightly ahead, using some of the topics covered in chapter 14. The example may be skipped, though, without loss of continuity.

A basic exception handling class can be constructed from which specific exception types are derived. Suppose we have a class Exception, having a member function Exception::severity. This member function tells us (little wonder!) the severity of a thrown exception. It might be Info, Notice, Warning, Error or Fatal. The information contained in the exception depends on its severity and is processed by a function handle. In addition, all exceptions support a member function like textMsg, returning textual information about the exception in a string.

By defining a polymorphic function handle it can be made to behave differently, depending on the nature of a thrown exception, when called from a basic Exception pointer or reference.
In this case, a program may throw any of these five exception types. Assuming that the classes `Message` and `Warning` were derived from the class `Exception`, then the `handle` function matching the exception type will automatically be called by the following exception catcher:

```cpp
//
catch(Exception &ex)
{
    cout << e.textMsg() << 'n';

    if
    {
        ex.severity() != ExceptionType::Warning
        &&
        ex.severity() != ExceptionType::Message
    }
    throw; // Pass on other types of Exceptions

    ex.handle(); // Process a message or a warning
}
```

Now anywhere in the `try` block preceding the exception handler, `Exception` objects or objects of one of its derived classes may be thrown. All those exceptions will be caught by the above handler. E.g.,

```cpp
throw Info();
throw Warning();
throw Notice();
throw Error();
throw Fatal();
```

## 10.4 The try block

The `try`-block surrounds `throw` statements. Remember that a program is always surrounded by a global `try` block, so `throw` statements may appear anywhere in your code. More often, though, `throw` statements are used in function bodies and such functions may be called from within `try` blocks.

A `try` block is defined by the keyword `try` followed by a compound statement. This block, in turn, must be followed by at least one `catch` handler:

```cpp
try
{
    // any statements here
}
catch(...) // at least one catch clause here
{}
```

`Try`-blocks are commonly nested, creating exception levels. For example, `main`'s code is surrounded by a `try`-block, forming an outer level handling exceptions. Within `main`'s `try`-block functions are called which may also contain `try`-blocks, forming the next exception level. As we have seen (section 10.3.1), exceptions thrown in inner level `try`-blocks may or may not be processed at that level. By
10.5. CATCHING EXCEPTIONS

10.5 Catching exceptions

A catch clause consists of the keyword catch followed by a parameter list defining one parameter specifying type and (parameter) name of the exception caught by that particular catch handler. This name may then be used as a variable in the compound statement following the catch clause. Example:

```cpp
catch (string &message)
{
    // code to handle the message
}
```

Primitive types and objects may be thrown as exceptions. It’s a bad idea to throw a pointer or reference to a local object, but a pointer to a dynamically allocated object may be thrown if the exception handler deletes the allocated memory to prevent a memory leak. Nevertheless, throwing such a pointer is dangerous as the exception handler won’t be able to distinguish dynamically allocated memory from non-dynamically allocated memory, as illustrated by the next example:

```cpp
try
{
    static int x;
    int *xp = &x;

    if (condition1)
        throw xp;

    xp = new int(0);
    if (condition2)
        throw xp;
}
catch (int *ptr)
{
    // delete ptr or not?
}
```

Close attention should be paid to the nature of the parameter of the exception handler, to make sure that when pointers to dynamically allocated memory are thrown the memory is returned once the handler has processed the pointer. In general pointers should not be thrown as exceptions. If dynamically allocated memory must be passed to an exception handler then the pointer should be wrapped in a smart pointer, like `unique_ptr` or `shared_ptr` (cf. sections 18.3 and 18.4).

Multiple catch handlers may follow a try block, each handler defining its own exception type. The order of the exception handlers is important. When an exception is thrown, the first exception handler matching the type of the thrown exception is used and remaining exception handlers are ignored. Eventually at most one exception handler following a try-block is activated. Normally this is of no concern as each exception has its own unique type.

Example: if exception handlers are defined for `char *`s and `void *`s then NTB strings are caught by the former handler. Note that a `char *` can also be considered a `void *`, but the exception type
matching procedure is smart enough to use the char * handler with the thrown NTBS. Handlers should be designed very type specific to catch the correspondingly typed exception. For example, int-exceptions are not caught by double-catchers, char-exceptions are not caught by int-catchers. Here is a little example illustrating that the order of the catchers is not important for types not having any hierarchal relationship to each other (i.e., int is not derived from double; string is not derived from an NTBS):

```cpp
#include <iostream>
using namespace std;

int main()
{
    while (true)
    {
        try
        {
            string s;
            cout << "Enter a,c,i,s for ascii-z, char, int, string "
                 "exception\n";
            getline(cin, s);
            switch (s[0])
            {
                case 'a':
                    throw "ascii-z";
                case 'c':
                    throw 'c';
                case 'i':
                    throw 12;
                case 's':
                    throw string();
            }
        }
        catch (string const &)
        {
            cout << "string caught\n";
        }
        catch (char const *)
        {
            cout << "ASCII-Z string caught\n";
        }
        catch (double)
        {
            cout << "isn’t caught at all\n";
        }
        catch (int)
        {
            cout << "int caught\n";
        }
        catch (char)
        {
            cout << "char caught\n";
        }
    }
}``
10.5. CATCHING EXCEPTIONS

Rather than defining specific exception handlers a specific class can be designed whose objects contain information about the exception. Such an approach was mentioned earlier, in section 10.3.1. Using this approach, there’s only one handler required, since we know we don’t throw other types of exceptions:

```cpp
try
{
    // code throws only Exception pointers
} catch (Exception &ex)
{
    ex.handle();
}
```

When the code of an exception handler has been processed, execution continues beyond the last exception handler directly following the matching try-block (assuming the handler doesn’t itself use flow control statements (like return or throw) to break the default flow of execution). The following cases can be distinguished:

- If no exception was thrown within the try-block no exception handler is activated, and execution continues from the last statement in the try-block to the first statement beyond the last catch-block.
- If an exception was thrown within the try-block but neither the current level nor another level contains an appropriate exception handler, the program’s default exception handler is called, aborting the program.
- If an exception was thrown from the try-block and an appropriate exception handler is available, then the code of that exception handler is executed. Following that, the program’s execution continues at the first statement beyond the last catch-block.

All statements in a try block following an executed throw-statement are ignored. However, objects that were successfully constructed within the try block before executing the throw statement are destroyed before any exception handler’s code is executed.

10.5.1 The default catcher

At a certain level of the program only a limited set of handlers may actually be required. Exceptions whose types belong to that limited set are processed, all other exceptions are passed on to exception handlers of an outer level try block.

An intermediate type of exception handling may be implemented using the default exception handler, which must be (due to the hierarchal nature of exception catchers, discussed in section 10.5) placed beyond all other, more specific exception handlers.

This default exception handler cannot determine the actual type of the thrown exception and cannot determine the exception’s value but it may execute some statements, and thus do some default processing. Moreover, the caught exception is not lost, and the default exception handler may use the empty throw statement (see section 10.3.1) to pass the exception on to an outer level, where it’s actually processed. Here is an example showing this use of a default exception handler:

```cpp
#include <iostream>
using namespace std;
```
int main() {
    try {
        try {
            throw 12.25; // no specific handler for doubles
        }
        catch (int value) {
            cout << "Inner level: caught int\n";
        }
        catch (...) {
            cout << "Inner level: generic handling of exceptions\n";
            throw;
        }
    }
    catch (double d) {
        cout << "Outer level may use the thrown double: " << d << '\n';
    }
}

/*
Generated output:
Inner level: generic handling of exceptions
Outer level may use the thrown double: 12.25
*/

The program's output illustrates that an empty `throw` statement in a default exception handler throws the received exception to the next (outer) level of exception catchers, keeping type and value of the thrown exception.

Thus, basic or generic exception handling can be accomplished at an inner level, while specific handling, based on the type of the thrown expression, can be provided at an outer level. Additionally, particularly in multi-threaded programs (cf. chapter 20), thrown exceptions can be transferred between threads after converting `std::exception` objects to `std::exception_ptr` objects. This procedure can even be used from inside the default catcher. Refer to section 20.13.1 for further coverage of the class `std::exception_ptr`.

### 10.6 Declaring exception throwers (deprecated)

Functions defined elsewhere may be linked to code that uses these functions. Such functions are normally declared in header files, either as standalone functions or as class member functions.

Those functions may of course throw exceptions. Declarations of such functions may contain a (now deprecated, see also section 23.7) function throw list or exception specification list specifying the types of the exceptions that can be thrown by the function. For example, a function that may throw 'char *' and 'int' exceptions can be declared as

```cpp
void exceptionThrower() throw(char *, int);
```
A function throw list immediately follows the function header (and it also follows a possible `const` specifier). Throw lists may be empty. It has the following general form:

\[
\text{throw}([\text{type1}, \text{type2}, \text{type3}, \ldots])
\]

If a function is guaranteed not to throw exceptions an empty function throw list may be used. E.g.,

\[
\text{void noExceptions()} \text{ throw } ();
\]

In all cases, the function header used in the function definition must exactly match the function header used in the declaration, including a possibly empty function throw list.

A function for which a function throw list is specified may only throw exceptions of the types mentioned in its throw list. A run-time error occurs if it throws other types of exceptions than those mentioned in the function throw list. Example: the function `charPintThrower` shown below clearly throws a `char const *` exception. Since `intThrower` may throw an `int` exception, the function throw list of `charPintThrower` must also contain `int`.

```cpp
#include <iostream>
using namespace std;

void charPintThrower() throw(char const *, int);

class Thrower
{
    public:
        void intThrower(int) const throw(int);
    
};

void Thrower::intThrower(int x) const throw(int)
{
    if (x)
        throw x;
}

void charPintThrower() throw(char const *, int)
{
    int x;
    
    cerr << "Enter an int: ";
    cin >> x;
    
    Thrower().intThrower(x);
    throw "this text is thrown if 0 was entered";
}

void runTimeError() throw(int)
{
    throw 12.5;
}

int main()
{

A function without a throw list may throw any kind of exception. Without a function throw list the program's designer is responsible for providing the correct handlers.

For various reason declaring exception throwers is now deprecated. Declaring exception throwers does not imply that the compiler checks whether an improper exception is thrown. Rather, the function will be surrounded by additional code in which the actual exception that is thrown is processed. Instead of compile time checks one gets run-time overhead, resulting in additional code (and execution time) that is added to the function's code. One could write, e.g.,

```cpp
void fun() throw (int)
{
    // code of this function, throwing exceptions
}
```

but the function would be compiled to something like the following (cf. section 10.11 for the use of `try` immediately following the function's header and section 10.8 for a description of bad_exception):

```cpp
void fun()
try  // this code resulting from throw(int)
{
    // the function’s code, throwing all kinds of exceptions
} catch (int) // remaining code resulting from throw(int)
{
    throw; // rethrow the exception, so it can be caught by the
    // ‘intended’ handler
} catch (...) // catch any other exception
{
    throw bad_exception;
```
Run-time overhead is caused by doubling the number of thrown and caught exceptions. Without a throw list a thrown int is simply caught by its intended handler; with a throw list the int is first caught by the ‘safeguarding’ handler added to the function. In there it is rethrown to be caught by its intended handler next.

10.7 Iostreams and exceptions

The C++ I/O library was used well before exceptions were available in C++. Hence, normally the classes of the iostream library do not throw exceptions. However, it is possible to modify that behavior using the ios::exceptions member function. This function has two overloaded versions:

- **ios::iostate exceptions()**:
  
  this member returns the state flags for which the stream will throw exceptions;

- **void exceptions(ios::iostate state)**
  
  this member causes the stream to throw an exception when state state is observed.

In the I/O library, exceptions are objects of the class ios::failure, derived from ios::exception. A `std::string const &message` may be specified when defining a failure object. Its message may then be retrieved using its virtual `char const *what() const` member.

Exceptions should be used in exceptional circumstances. Therefore, we think it is questionable to have stream objects throw exceptions for fairly normal situations like EOF. Using exceptions to handle input errors might be defensible (e.g., in situations where input errors should not occur and imply a corrupted file) but often aborting the program with an appropriate error message would probably be the more appropriate action. As an example consider the following interactive program using exceptions to catch incorrect input:

```cpp
#include <iostream>
#include <climits>
using namespace::std;

int main()
{
    cin.exceptions(ios::failbit); // throw exception on fail
    while (true)
    {
        try
        {
            cout << "enter a number: ";
            int value;
            cin >> value;
            cout << "you entered " << value << ‘\n’;
        }
        catch (ios::failure const &problem)
        {
            cout << problem.what() << ‘\n’;
        }
    }
    return 0;
}
```
cin.clear();
cin.ignore(INT_MAX, '\n'); // ignore the faulty line
}
}
}

By default, exceptions raised from within ostream objects are caught by these objects, which set their ios::badbit as a result. See also the paragraph on this issue in section 14.8.

10.8 Standard Exceptions

All data types may be thrown as exceptions. Several additional exception classes are now defined by the C++ standard. Before using those additional exception classes the <stdexcept> header file must be included. All of these standard exceptions are class types by themselves, but also offer all facilities of the std::exception class and objects of the standard exception classes may also be considered objects of the std::exception class.

The std::exception class offers the member

    char const *what() const;

describing in a short textual message the nature of the exception.

C++ defines the following standard exception classes:

- std::bad_alloc (this requires the <new> header file): thrown when operator new fails;
- std::bad_exception (this requires the header file <exception> header file): thrown when a function tries to generate another type of exception than declared in its function throw list;
- std::bad_cast (this requires the <typeinfo> header file): thrown in the context of polymorphism (see section 14.6.1);
- std::bad_typeid (this requires the <typeinfo> header file): also thrown in the context of polymorphism (see section 14.6.2);

All additional exception classes were derived from std::exception. The constructors of all these additional classes accept std::string const & arguments summarizing the reason for the exception (retrieved by the exception::what member). The additionally defined exception classes are:

- std::domain_error: a (mathematical) domain error is detected;
- std::invalid_argument: the argument of a function has an invalid value;
- std::length_error: thrown when an object would have exceeded its maximum permitted length;
- std::logic_error: a logic error should be thrown when a problem is detected in the internal logic of the program. Example: a function like C’s printf is called with more arguments than there are format specifiers in its format string;
- std::out_of_range: thrown when an argument exceeds its permitted range. Example: thrown by at members when their arguments exceed the range of admissible index values;
• `std::overflow_error`: an overflow error should be thrown when an arithmetic overflow is detected. Example: dividing a value by a very small value;

• `std::range_error`: a range error should be thrown when an internal computation results in a value exceeding a permissible range;

• `std::runtime_error`: a runtime error should be thrown when a problem is encountered that can only be detected while the program is being executed. Example: a non-integral is entered when the program’s input expects an integral value.

• `std::underflow_error`: an underflow error should be thrown when an arithmetic underflow is detected. Example: dividing a very small value by a very large value.

### 10.9 System error, error code and error category

A `std::system_error` can be thrown when an error occurs that has an associated error code. Such errors are typically encountered when calling low-level (like operating system) functions.

Before using `system_error` the `<system_error>` header file must be included.

A `system_error` object can be constructed using the standard textual description of the nature of the encountered error, but in addition accepts an `error_code` or `error_category` object (see the next two sections), further specifying the nature of the error. The `error_code` and `error_category` classes are also declared in the `system_error` header file.

The header file `system_error` also defines an enum class `errc` whose values are equal to and describe in a less cryptic way the traditional error code values as offered by C macros, e.g.,

```cpp
enum class errc
{
    address_family_not_supported, // EAFNSUPPORT
    address_in_use, // EADDRINUSE
    address_not_available, // EADDRNOTAVAIL
    already_connected, // EISCONN
    argument_list_too_long, // E2BIG
    argument_out_of_domain, // EDOM
    bad_address, //EFAULT
    ...
};
```

In addition to the standard `what` member, the `system_error` class also offers a member `code` returning a const reference to the exception’s error code. Here is the class’s public interface:

```cpp
class system_error: public runtime_error
{
    public:
        system_error(error_code ec, string const &what_arg);
        system_error(error_code ec, char const *what_arg);
        system_error(error_code ec);
        system_error(int ev, error_category const &ecat,
                      string const &what_arg);
        system_error(int ev, error_category const &ecat,
                      char const *what_arg);
};
```
system_error(int ev, error_category const &ecat);
error_code const &code() const noexcept;
char const *what() const noexcept;

The NTBS returned by its `what` member may be formatted by a `system_error` object like this:

what_arg + ": " + code().message()

Note that, although `system_error` was derived from `runtime_error`, you’ll lose the `code` member when catching a `std::exception` object. Of course, downcasting is always possible, but that’s a stopgap. Therefore, if a `system_error` is thrown, a matching `catch(system_error const &)` clause should be provided (for a flexible alternative, see the class `FBB::Exception` in the author’s Bobcat library¹.

10.9.1 The class ‘std::error_code’

Objects of the class `std::error_code` hold `error code values`, which may be defined by the operating system or comparable low-level functions.

Before using `error_code` the `<system_error>` header file must be included.

The class offers the following constructors, members, and free functions:

Constructors:

- `error_code() noexcept`:
  the default construction initializes the `error` code with an `error value` 0 and an `error category` set to `&system_category();`

- `error_code(ErrorCodeEnum e) noexcept`:
  this is a member template (cf. section 22.1.3), defining template <class ErrorCodeEnum>. It initializes the object with the return value of `make_error_code(e)`.

The copy constructor is also available.

Members:

- `void assign(int val, const error_category& cat)`:
  assigns new values to the current object’s `value` and `category` data members;

- `error_category const &category() const noexcept`:
  returns a reference to the object’s `error category`;

- `void clear() noexcept`:
  after calling this member `value` is set to 0 and the object’s `error category` set to `&system_category();`

¹http://bobcat.sourceforge.net
• error_condition default_error_condition() const noexcept:
  returns category().default_error_condition(value());

• string message() const:
  returns category().message(value());

• errorcode& operator=(ErrorCodeEnum e) noexcept:
  a member template defining template <class ErrorCodeEnum>. It assigns the
  return value of make_error_code(e) to the current object;

• explicit operator bool() const noexcept:
  returns value() != 0;

• int value() const noexcept:
  returns the object's error value.

Free functions:

• error_code make_error_code(errc e) noexcept:
  returns error_code(static_cast<int>(e), generic_category());

• bool operator<(error_code const &lhs, error_code const &rhs) noexcept:
  returns
  lhs.category() < rhs.category()
  ||
  lhs.category() == rhs.category() && lhs.value() < rhs.value();

• std::ostream &operator«(std::ostream & os, error_code const &ec):
  inserts the following text into os:
  os << ec.category().name() << ':' << ec.value().

10.9.2 The class ‘std::error_category’

The class std::error_category serves as a base class for types used to identify the source and
encoding of a particular category of error code.

Before using error_category the <system_error> header file must be included.

Classes that are derived from error_category to support categories of errors in addition to those
defined by the C++11 standard. Other than that, the behavior of such derived classes should differ
from the behavior of the error_category class itself. Moreover, such derived classes should not
alter errno's value, or error states provided by other libraries.

The equality of error_category objects is deducted from the equality of their addresses. As
error_category objects are passed by reference, programs using objects of classes derived from
error_category should ensure that only a single object of each such type is actually used: the class
is designed as a Singleton (cf. Singleton Design Pattern (cf. Gamma et al. (1995) Design Patterns,
Addison-Wesley)): looking at the class's public interface it becomes clear that no error_category
object can immediately be constructed. There is no public constructor. Nor is it possible to copy
an existing `error_category` object, as the copy constructor and overloaded assignment operators have been deleted. Derived classes should enforce these singleton characteristics as well. Here is the `error_category`'s non-private class interface:

```cpp
class error_category
{
public:
    error_category(error_category const &) = delete;
    virtual ~error_category() noexcept;
    error_category& operator=(error_category const &) = delete;

    virtual char const *name() const noexcept = 0;
    virtual string message(int ev) const = 0;

    virtual error_condition
    default_error_condition(int ev) const noexcept;

    virtual bool equivalent(int code, error_condition const &condition)
    const noexcept;

    bool operator==(error_category const &rhs) const noexcept;
    bool operator!=(error_category const &rhs) const noexcept;
    bool operator<(error_category const &rhs) const noexcept;

protected:
    error_category() noexcept;
};
```

Members:

- `char const *name()` const noexcept:
  
  must be overridden, and should return a textual name of the error category;

- `string message(int ev) const`:
  
  must be overridden, and should return a string describing the error condition denoted by `ev`;

- `error_condition default_error_condition(int ev) const noexcept`:
  
  returns `error_condition(ev, *this)` (An object of type `error_condition` that corresponds to `ev`);

- `bool equivalent(int code, error_condition const &condition) const noexcept`:
  
  returns `default_error_condition(code) == condition` (true if, for the category of error represented by *this, code is considered equivalent to condition; otherwise false);
10.10. EXCEPTION GUARANTEES

- **bool equivalent(error_code const &code, int condition) const noexcept:**
  
  returns *this == code.category() && code.value() == condition (true if, for the category of error represented by *this, code is considered equivalent to condition; otherwise false);

- **bool operator<(error_category const &rhs) const noexcept:**
  
  returns less<const error_category*>()(this, &rhs).

Free functions:

- **error_category const &generic_category() noexcept:**
  
  returns a reference to an object of a type derived from the class error_category. Since error_category and its derived classes should be singleton classes, calls to this function must return references to the same object. The returned object's name member shall return a pointer to the string "generic";

- **error_category const &system_category() noexcept:**
  
  returns a reference to an object of a type derived from the class error_category. Since error_category and its derived classes should be singleton classes, calls to this function must return references to the same object. The object's name member shall return a pointer to the string "system". If the argument ev corresponds to a POSIX errno value 'posv', then the object's default_error_condition member should return error-condition(posv, generic_category()). Otherwise, error_condition(ev, system_category()) shall be returned.

10.10 Exception guarantees

Software should be **exception safe**: the program should continue to work according to its specifications in the face of exceptions. It is not always easy to realize exception safety. In this section some guidelines and terminology is introduced when discussing exception safety.

Since exceptions may be generated from within all C++ functions, exceptions may be generated in many situations. Not all of these situations are immediately and intuitively recognized as situations where exceptions can be thrown. Consider the following function and ask yourself at which points exceptions may be thrown:

```cpp
void fun()
{
    X x;
    cout << x;
    X *xp = new X(x);
    cout << (x + *xp);
    delete xp;
}
```

If it can be assumed that cout as used above does not throw an exception there are at least 13 opportunities for exceptions to be thrown:

- **X x**: the default constructor could throw an exception (#1)
• `cout << x`: the overloaded insertion operator could throw an exception (#2), but its rvalue argument might not be an `X` but, e.g., an `int`, and so `X::operator int() const` could be called which offers yet another opportunity for an exception (#3).

• `*xp = new X(x)`: the copy constructor may throw an exception (#4) and `operator new` (#5a) too. But did you realize that this latter exception might not be thrown from `::new`, but from, e.g., `X`'s own overload of `operator new`? (#5b)

• `cout << (x + *xp)`: we might be seduced into thinking that two `X` objects are added. But it doesn’t have to be that way. A separate class `Y` might exist and `X` may have a conversion operator `operator Y() const`, and `operator+(Y const &lhs, X const &rhs)`, `operator+(X const &lhs, Y const &rhs)`, and `operator+(X const &lhs, X const &rhs)` might all exist. So, if the conversion operator exists, then depending on the kind of overload of `operator+` that is defined either the addition’s left-hand side operand (#6), right-hand side operand (#7), or `operator+` itself (#8) may throw an exception. The resulting value may again be of any type and so the overloaded `cout << return-type-of-operator+` operator may throw an exception (#9). Since `operator+` returns a temporary object it is destroyed shortly after its use. `X`’s destructor could throw an exception (#10).

• `delete xp`: whenever `operator new` is overloaded `operator delete` should be overloaded as well and may throw an exception (#11). And of course, `X`’s destructor might again throw an exception (#12).

• `x`: when the function terminates the local `x` object is destroyed: again an exception could be thrown (#13).

It is stressed here (and further discussed in section 10.12) that although it is possible for exceptions to leave destructors this would violate the C++ standard and so it must be prevented in well-behaving C++ programs.

How can we expect to create working programs when exceptions might be thrown at this many situations?

Exceptions may be generated in a great many situations, but serious problems are prevented when we’re able to provide at least one of the following exception guarantees:

• **The basic guarantee**: no resources are leaked. In practice this means: all allocated memory is properly returned when exceptions are thrown.

• **The strong guarantee**: the program’s state remains unaltered when an exception is thrown (as an example: the canonical form of the overloaded assignment operator provides this guarantee)

• **The nothrow guarantee**: this applies to code for which it can be proven that no exception can be thrown from it.

### 10.10.1 The basic guarantee

The basic guarantee dictates that functions that fail to complete their assigned tasks must return all allocated resources, usually memory, before terminating. Since practically all functions and operators may throw exceptions and since a function may repeatedly allocate resources the blueprint of a function allocating resources shown below defines a try block to catch all exceptions that might be thrown. The catch handler’s task is to return all allocated resources and then rethrow the exception.

```c++
void allocator(X **xDest, Y **yDest) {
```
In the pre-try code the pointers to receive the addresses returned by the operator `new` calls are initialized to 0. Since the catch handler must be able to return allocated memory they must be available outside of the `try` block. If the allocation succeeds the memory pointed to by the destination pointers is returned and then the pointers are given new values.

Allocation and or initialization might fail. If allocation fails `new` throws a `std::bad_alloc` exception and the catch handler simply deletes 0 pointers which is OK.

If allocation succeeds but the construction of (some) of the objects fails by throwing an exception then the following is guaranteed to happen:

- The destructors of all successfully allocated objects are called;
- The dynamically allocated memory to contain the objects is returned

Consequently, there is no memory leak when `new` fails. Inside the above `try` block `new` `X` may fail: this does not affect the 0-pointers and so the catch handler merely deletes 0 pointers. When `new` `Y` fails `xp` points to allocated memory and so it must be returned. This happens inside the catch handler. The final pointer (here: `yp`) will only be unequal zero when `new` `Y` properly completes, so there’s no need for the catch handler to return the memory pointed at by `yp`.

### 10.10.2 The strong guarantee

The strong guarantee dictates that an object’s state should not change in the face of exceptions. This is realized by performing all operations that might throw on a separate copy of the data. If all this succeeds then the current object and its (now successfully modified) copy are swapped. An example of this approach can be observed in the canonical overloaded assignment operator:

```cpp
Class &operator=(Class const &other)
{
    Class tmp(other);
    swap(tmp);
    return *this;
}
```
The copy construction might throw an exception, but this keeps the current object’s state intact. If the copy construction succeeds swap swaps the current object’s contents with tmp’s contents and returns a reference to the current object. For this to succeed it must be guaranteed that swap won’t throw an exception. Returning a reference (or a value of a primitive data type) is also guaranteed not to throw exceptions. The canonical form of the overloaded assignment operator therefore meets the requirements of the strong guarantee.

Some rules of thumb were formulated that relate to the strong guarantee (cf. Sutter, H., *Exceptional C++*, Addison-Wesley, 2000). E.g.,

- All the code that might throw an exception affecting the current state of an object should perform its tasks separately from the data controlled by the object. Once this code has performed its tasks without throwing an exception replace the object’s data by the new data.
- Member functions modifying their object’s data should not return original (contained) objects by value.

The canonical assignment operator is a good example of the first rule of thumb. Another example is found in classes storing objects. Consider a class PersonDb storing multiple Person objects. Such a class might offer a member void add(Person const &next). A plain implementation of this function (merely intended to show the application of the first rule of thumb, but otherwise completely disregarding efficiency considerations) might be:

```cpp
void PersonDb::newAppend(Person const &next)
{
    Person *tmp = 0;
    try
    {
        tmp = new Person[d_size + 1];
        for (size_t idx = 0; idx < d_size; ++idx)
            tmp[idx] = d_data[idx];
        tmp[d_size] = next;
    }
    catch (...)
    {
        delete[] tmp;
        throw;
    }
}

void PersonDb::add(Person const &next)
{
    Person *tmp = newAppend(next);
    delete[] d_data;
    d_data = tmp;
    ++d_size;
}
```

The (private) newAppend member’s task is to create a copy of the currently allocated Person objects, including the data of the next Person object. Its catch handler catches any exception that might be thrown during the allocation or copy process and returns all memory allocated so far, rethrowing the exception at the end. The function is exception neutral as it propagates all its exceptions to its
10.10. EXCEPTION GUARANTEES

The function also doesn’t modify the PersonDb object’s data, so it meets the strong exception guarantee. Returning from newAppend the member add may now modify its data. Its existing data are returned and its d_data pointer is made to point to the newly created array of Person objects. Finally its d_size is incremented. As these three steps don't throw exceptions add too meets the strong guarantee.

The second rule of thumb (member functions modifying their object's data should not return original (contained) objects by value) may be illustrated using a member PersonDb::erase(size_t idx). Here is an implementation attempting to return the original d_data[idx] object:

```cpp
Person PersonData::erase(size_t idx)
{
    if (idx >= d_size)
        throw string("Array bounds exceeded");
    Person ret(d_data[idx]);
    Person *tmp = copyAllBut(idx);
    delete[] d_data;
    d_data = tmp;
    --d_size;
    return ret;
}
```

Although copy elision usually prevents the use of the copy constructor when returning ret, this is not guaranteed to happen. Furthermore, a copy constructor may throw an exception. If that happens the function has irrevocably mutated the PersonDb's data, thus losing the strong guarantee.

Rather than returning d_data[idx] by value it might be assigned to an external Person object before mutating PersonDb's data:

```cpp
void PersonData::erase(Person *dest, size_t idx)
{
    if (idx >= d_size)
        throw string("Array bounds exceeded");
    *dest = d_data[idx];
    Person *tmp = copyAllBut(idx);
    delete[] d_data;
    d_data = tmp;
    --d_size;
}
```

This modification works, but changes the original assignment of creating a member returning the original object. However, both functions suffer from a task overload as they modify PersonDb’s data and also return an original object. In situations like these the one-function-one-responsibility rule of thumb should be kept in mind: a function should have a single, well defined responsibility.

The preferred approach is to retrieve PersonDb’s objects using a member like Person const &at(size_t idx) const and to erase an object using a member like void PersonData::erase(size_t idx).

### 10.10.3 The nothrow guarantee

Exception safety can only be realized if some functions and operations are guaranteed not to throw exceptions. This is called the nothrow guarantee. An example of a function that must offer the
nothrow guarantee is the swap function. Consider once again the canonical overloaded assignment operator:

```cpp
Class &operator=(Class const &other)
{
    Class tmp(other);
    swap(tmp);
    return *this;
}
```

If swap were allowed to throw exceptions then it would most likely leave the current object in a partially swapped state. As a result the current object's state would most likely have been changed. As tmp has been destroyed by the time a catch handler receives the thrown exception it becomes very difficult (as in: impossible) to retrieve the object's original state. Losing the strong guarantee as a consequence.

The swap function must therefore offer the nothrow guarantee. It must have been designed as if using the following prototype (see also section 23.7):

```cpp
void Class::swap(Class &other) noexcept;
```

Likewise, operator delete and operator delete[] offer the nothrow guarantee, and according to the C++ standard destructors may themselves not throw exceptions (if they do their behavior is formally undefined, see also section 10.12 below).

Since the C programming language does not define the exception concept functions from the standard C library offer the nothrow guarantee by implication. This allowed us to define the generic swap function in section 9.6 using memcpy.

Operations on primitive types offer the nothrow guarantee. Pointers may be reassigned, references may be returned etc. etc. without having to worry about exceptions that might be thrown.

### 10.11 Function try blocks

Exceptions may be generated while a constructor is initializing its members. How can exceptions generated in such situations be caught by the constructor itself, rather than outside the constructor? The intuitive solution, nesting the object construction in a try block does not solve the problem. The exception by then has left the constructor and the object we intended to construct isn't visible anymore.

Using a nested try block is illustrated in the next example, where main defines an object of class PersonDb. Assuming that PersonDb's constructor throws an exception, there is no way we can access the resources that might have been allocated by PersonDb's constructor from the catch handler as the pdb object is out of scope:

```cpp
int main(int argc, char **argv)
{
    try
    {
        PersonDb pdb(argc, argv); // may throw exceptions
        ... // main()'s other code
    }
...
catch(...) // and/or other handlers
{
    ...
    // pdb is inaccessible from here
}

Although all objects and variables defined inside a try block are inaccessible from its associated catch handlers, object data members were available before starting the try block and so they may be accessed from a catch handler. In the following example the catch handler in PersonDb's constructor is able to access its d_data member:

```cpp
PersonDb::PersonDb(int argc, char **argv) :
    d_data(0),
    d_size(0)
{
    try
    {
        initialize(argc, argv);
    }
    catch(...)
    {
        // d_data, d_size: accessible
    }
}
```

Unfortunately, this does not help us much. The initialize member is unable to reassign d_data and d_size if PersonDb const pdb was defined; the initialize member should at least offer the basic exception guarantee and return any resources it has acquired before terminating due to a thrown exception; and although d_data and d_size offer the nothrow guarantee as they are of primitive data types a class type data member might throw an exception, possibly resulting in violation of the basic guarantee.

In the next implementation of PersonDb assume that constructor receives a pointer to an already allocated block of Person objects. The PersonDb object takes ownership of the allocated memory and it is therefore responsible for the allocated memory's eventual destruction. Moreover, d_data and d_size are also used by a composed object PersonDbSupport, having a constructor expecting a Person const * and size_t argument. Our next implementation may then look something like this:

```cpp
PersonDb::PersonDb(Person *pData, size_t size) :
    d_data(pData),
    d_size(size),
    d_support(d_data, d_size)
{
    // no further actions
}
```

This setup allows us to define a PersonDb const &pdb. Unfortunately, PersonDb cannot offer the basic guarantee. If PersonDbSupport's constructor throws an exception it isn't caught although d_data already points to allocated memory.

The function try block offers a solution for this problem. A function try block consists of a try block and its associated handlers. The function try block starts immediately after the function
header, and its block defines the function body. With constructors base class and data member initializers may be placed between the try keyword and the opening curly brace. Here is our final implementation of PersonDb, now offering the basic guarantee:

```cpp
PersonDb::PersonDb(Person *pData, size_t size)
try :
    d_data(pData),
    d_size(size),
    d_support(d_data, d_size)
{} 
catch (...) 
{
    delete[] d_data;
}
```

Let's have a look at a stripped-down example. A constructor defines a function try block. The exception thrown by the Throw object is initially caught by the object itself. Then it is rethrown. The surrounding Composer's constructor also defines a function try block, Throw's rethrown exception is properly caught by Composer's exception handler, even though the exception was generated from within its member initializer list:

```cpp
#include <iostream>

class Throw
{
    public:
    Throw(int value)
    try 
    {
        throw value;
    }
    catch(...) 
    {
        std::cout << "Throw's exception handled locally by Throw()\n";
        throw;
    }
};

class Composer
{
    Throw d_t;
    public:
    Composer() // NOTE: try precedes initializer list
    try 
    {
        d_t(5)
    }
    catch(...) 
    {
        std::cout << "Composer() caught exception as well\n";
    }
};
```
int main()
{
    Composer c;
}

When running this example, we’re in for a nasty surprise: the program runs and then breaks with an *abort exception*. Here is the output it produces, the last two lines being added by the system’s final catch-all handler, catching all remaining uncaught exceptions:

```
Throw’s exception handled locally by Throw()
Composer() caught exception as well
terminate called after throwing an instance of ‘int’
Abort
```

The reason for this is documented in the C++ standard: at the end of a catch-handler belonging to a constructor or destructor function try block, the original exception is automatically rethrown.

The exception is not rethrown if the handler itself throws another exception, offering the constructor or destructor a way to replace a thrown exception by another one. The exception is only rethrown if it reaches the end of the catch handler of a constructor or destructor function try block. Exceptions caught by nested catch handlers are not automatically rethrown.

As only constructors and destructors rethrow exceptions caught by their function try block catch handlers the run-time error encountered in the above example may simply be repaired by providing `main` with its own function try block:

```
int main()
try
{
    Composer c;
}
catch (...)
{}
```

Now the program runs as planned, producing the following output:

```
Throw’s exception handled locally by Throw()
Composer() caught exception as well
```

A final note: if a function defining a function try block also declares an exception throw list then only the types of rethrown exceptions must match the types mentioned in the throw list.

### 10.12 Exceptions in constructors and destructors

Object destructors are only activated for completely constructed objects. Although this may sound like a truism, there is a subtlety here. If the construction of an object fails for some reason, the object’s destructor is *not* called when the object goes out of scope. This could happen if an exception that is generated by the constructor is not caught by the constructor. If the exception is thrown when the object has already allocated some memory, then that memory is not returned: its destructor isn’t called as the object’s construction wasn’t successfully completed.
The following example illustrates this situation in its prototypical form. The constructor of the class `Incomplete` first displays a message and then throws an exception. Its destructor also displays a message:

```cpp
class Incomplete {
public:
    Incomplete() {
        cerr << "Allocated some memory\n";
        throw 0;
    }

    ~Incomplete() {
        cerr << "Destroying the allocated memory\n";
    }
};
```

Next, `main()` creates an `Incomplete` object inside a `try` block. Any exception that may be generated is subsequently caught:

```cpp
int main() {
    try {
        cerr << "Creating 'Incomplete' object\n";
        Incomplete();
        cerr << "Object constructed\n";
    }
    catch(...) {
        cerr << "Caught exception\n";
    }
}
```

When this program is run, it produces the following output:

```
Creating 'Incomplete' object
Allocated some memory
Caught exception
```

Thus, if `Incomplete`'s constructor would actually have allocated some memory, the program would suffer from a memory leak. To prevent this from happening, the following counter measures are available:

- **Prevent the exceptions from leaving the constructor.**
  If part of the constructor's body may generate exceptions, then this part may be surrounded by a `try` block, allowing the exception to be caught by the constructor itself. This approach is defensible when the constructor is able to repair the cause of the exception and to complete its construction as a valid object.

- **If an exception is generated by a base class constructor or by a member initializing constructor then a `try` block within the constructor's body won't be able to catch the thrown exception.**
This *always* results in the exception leaving the constructor and the object is not considered to have been properly constructed. A *try* block may include the member initializers, and the *try* block's compound statement becomes the constructor's body as in the following example:

```cpp
class Incomplete2
{
    Composed d_composed;

public:

    Incomplete2()
    try
    {
        d_composed(/* arguments */)
    }
    catch (...)
    {
    }
};
```

An exception thrown by either the member initializers or the body results in the execution never reaching the body's closing curly brace. Instead the catch clause is reached. Since the constructor's body isn't properly completed the object is not considered properly constructed and eventually the object's destructor won't be called.

The catch clause of a constructor's function *try* block behaves slightly different than a catch clause of an ordinary function *try* block. An exception reaching a constructor's function *try* block may be transformed into another exception (which is thrown from the catch clause) but if no exception is explicitly thrown from the catch clause the exception originally reaching the catch clause is always rethrown. Consequently, there's no way to confine an exception thrown from a base class constructor or from a member initializer to the constructor: such an exception *always* propagates to a more shallow block and in that case the object's construction is always considered incomplete.

Consequently, if incompletely constructed objects throw exceptions then the constructor's catch clause is responsible for preventing memory (generally: resource) leaks. There are several ways to realize this:

- When multiple inheritance is used: if initial base classes have properly been constructed and a later base class throws, then the initial base class objects are automatically destroyed (as they are themselves fully constructed objects)
- When composition is used: already constructed composed objects are automatically destroyed (as they are fully constructed objects)
- Instead of using plain pointers *smart pointers* (cf. section 18.4) should be used to manage dynamically allocated memory. In this case, if the constructor throws either before or after the allocation of the dynamic memory, then allocated memory is properly returned as *shared_ptr* objects are, after all, objects.
- If plain pointer data members *must* be used then the constructor's body should first, in its member initialization section, initialize its plain pointer data members. Then, in its body it can dynamically allocate memory, reassigning the plain pointer data members. The constructor must be provided with a function *try* block whose generic catch clause deletes the memory pointed at by the class's plain pointer data members. Example:

```cpp
class Incomplete2
```
CHAPTER 10. EXCEPTIONS

```cpp
{ 
    Composed d_composed;
    char *d_cp; // plain pointers
    int *d_ip;

public:
    Incomplete2(size_t nChars, size_t nInts) 
    try :
    
    d_composed(/* arguments */), // might throw
    d_cp(0),
    d_ip(0)
    {
    preamble(); // might throw
    d_cp = new char[nChars]; // might throw
    d_ip = new int[nChars]; // might throw
    postamble(); // might throw
    
    catch (...) 
    {
    delete[] d_cp; // clean up
    delete[] d_ip;
    
    });

On the other hand, since C++11 offers constructor delegation an object may have been completely constructed according to the C++ run-time system, but yet its constructor may have thrown an exception. This happens if a delegated constructor successfully completes (after which the object is considered ‘completely constructed’), but the constructor itself throws an exception, as illustrated by the next example:

    class Delegate
    {
    public:
        Delegate()
        :
            Delegate(0)
        {
        throw 12; // throws but completely constructed
        }
    Delegate(int x) // completes OK
    {
    }
    
    int main()
    try
    {
    Delegate del; // throws
    
    } // del’s destructor is called here
    catch (...)
    {
    }

    In this example it is the responsibility of Delegate’s designer to ensure that the throwing default constructor does not invalidate the actions performed by Delegate’s destructor. E.g., if the del-
egated constructor allocates memory to be deleted by the destructor, then the default constructor
should either leave the memory as-is, or it can delete the memory and set the corresponding pointer
to zero thereafter. In any case, it is Delegate’s responsibility to ensure that the object remains in a
valid state, even though it throws an exception.

According to the C++ standard exceptions thrown by destructors may not leave their bodies. Providing
a destructor with a function try block is therefore a violation of the standard: exceptions caught
by a function try block’s catch clause have already left the destructor’s body. If—in violation of the
standard— the destructor is provided with a function try block and an exception is caught by the
try block then that exception is rethrown, similar to what happens in catch clauses of constructor
functions’ try blocks.

The consequences of an exception leaving the destructor’s body is not defined, and may result in
unexpected behavior. Consider the following example:

Assume a carpenter builds a cupboard having a single drawer. The cupboard is finished, and a
customer, buying the cupboard, finds that the cupboard can be used as expected. Satisfied with the
cupboard, the customer asks the carpenter to build another cupboard, this time having two drawers.
When the second cupboard is finished, the customer takes it home and is utterly amazed when the
second cupboard completely collapses immediately after it is used for the first time.

Weird story? Then consider the following program:

```c++
int main()
{
  try
  {
    cerr << "Creating Cupboard1\n";
    Cupboard1();
    cerr << "Beyond Cupboard1 object\n";
  }
  catch (...)
  {
    cerr << "Cupboard1 behaves as expected\n";
  }
  try
  {
    cerr << "Creating Cupboard2\n";
    Cupboard2();
    cerr << "Beyond Cupboard2 object\n";
  }
  catch (...)
  {
    cerr << "Cupboard2 behaves as expected\n";
  }
}
```

When this program is run it produces the following output:

```
Creating Cupboard1
Drawer 1 used
Cupboard1 behaves as expected
Creating Cupboard2
Drawer 2 used
Drawer 1 used
```
terminate called after throwing an instance of 'int'
  Abort

The final Abort indicates that the program has aborted instead of displaying a message like
Cupboard2 behaves as expected.

Let's have a look at the three classes involved. The class Drawer has no particular characteristics,
except that its destructor throws an exception:

class Drawer
{
    size_t d_nr;
    public:
        Drawer(size_t nr)
        :
            d_nr(nr)
        {}
        ~Drawer()
        {
            cerr << "Drawer " << d_nr << " used\n";
            throw 0;
        }
};

The class Cupboard1 has no special characteristics at all. It merely has a single composed Drawer object:

class Cupboard1
{
    Drawer left;
    public:
        Cupboard1()
        :
            left(1)
        {}
};

The class Cupboard2 is constructed comparably, but it has two composed Drawer objects:

class Cupboard2
{
    Drawer left;
    Drawer right;
    public:
        Cupboard2()
        :
            left(1),
            right(2)
        {}
};

When Cupboard1's destructor is called Drawer's destructor is eventually called to destroy its composed object. This destructor throws an exception, which is caught beyond the program's first try block. This behavior is completely as expected.
A subtlety here is that Cupboard1’s destructor (and hence Drawer’s destructor) is activated immediately subsequent to its construction. Its destructor is called immediately subsequent to its construction as Cupboard1() defines an anonymous object. As a result the Beyond Cupboard1 object text is never inserted into std::cerr.

Because of Drawer’s destructor throwing an exception a problem occurs when Cupboard2’s destructor is called. Of its two composed objects, the second Drawer’s destructor is called first. This destructor throws an exception, which ought to be caught beyond the program’s second try block. However, although the flow of control by then has left the context of Cupboard2’s destructor, that object hasn’t completely been destroyed yet as the destructor of its other (left) Drawer still has to be called.

Normally that would not be a big problem: once an exception is thrown from Cupboard2’s destructor any remaining actions would simply be ignored, albeit that (as both drawers are properly constructed objects) left’s destructor would still have to be called.

This happens here too and left’s destructor also needs to throw an exception. But as we’ve already left the context of the second try block, the current flow control is now thoroughly mixed up, and the program has no other option but to abort. It does so by calling terminate(), which in turn calls abort(). Here we have our collapsing cupboard having two drawers, even though the cupboard having one drawer behaves perfectly.

The program aborts since there are multiple composed objects whose destructors throw exceptions leaving the destructors. In this situation one of the composed objects would throw an exception by the time the program’s flow control has already left its proper context causing the program to abort.

The C++ standard therefore understandably stipulates that exceptions may never leave destructors. Here is the skeleton of a destructor whose code might throw exceptions. No function try block but all the destructor’s actions are encapsulated in a try block nested under the destructor’s body.

```cpp
Class::~Class()
{
    try
    {
        maybe.throw.exceptions();
    }
    catch (...) {}
}
```
Chapter 11

More Operator Overloading

Having covered the overloaded assignment operator in chapter 9, and having shown several examples of other overloaded operators as well (i.e., the insertion and extraction operators in chapters 3 and 6), we now take a look at operator overloading in general.

11.1 Overloading ‘operator[]()’

As our next example of operator overloading, we introduce a class `IntArray` encapsulating an array of `ints`. Indexing the array elements is possible using the standard array index operator [], but additionally checks for array bounds overflow are performed. Furthermore, the index operator (operator[]) is interesting in that it can be used in expressions as both lvalue and as rvalue.

Here is an example showing the basic use of the class:

```cpp
int main()
{
    IntArray x(20); // 20 ints
    for (int i = 0; i < 20; i++)
        x[i] = i * 2; // assign the elements

    for (int i = 0; i <= 20; i++) // produces boundary overflow
        cout << "At index " << i << ": value is " << x[i] << '\n';
}
```

First, the constructor is used to create an object containing 20 `ints`. The elements stored in the object can be assigned or retrieved. The first `for`-loop assigns values to the elements using the index operator, the second `for`-loop retrieves the values but also results in a run-time error once the non-existing value `x[20]` is addressed. The `IntArray` class interface is:

```cpp
#include <cstddef>

class IntArray
{
    int *d_data;
    size_t d_size;
};
```
This class has the following characteristics:

- One of its constructors has a `size_t` parameter having a default argument value, specifying the number of `int` elements in the object.

- The class internally uses a pointer to reach allocated memory. Hence, the necessary tools are provided: a copy constructor, an overloaded assignment operator and a destructor.

- Note that there are two overloaded index operators. Why are there two?

  The first overloaded index operator allows us to reach and modify the elements of non-constant `IntArray` objects. This overloaded operator's prototype is a function returning a reference to an `int`. This allows us to use expressions like `x[10]` as rvalues or lvalues. With non-const `IntArray` objects `operator[]` can therefore be used to retrieve and to assign values. The return value of the non-const `operator[]` member is not an `int const &`, but an `int &`. In this situation we don't use `const`, as we must be able to modify the element we want to access when the operator is used as lvalue.

  This whole scheme fails if there's nothing to assign. Consider the situation where we have an `IntArray const stable(5)`. Such an object is an immutable `const` object. The compiler detects this and refuses to compile this object definition if only the non-const `operator[]` is available. Hence the second overloaded index operator is added to the class's interface. Here the return value is an `int const &`, rather than an `int &`, and the member function itself is a `const` member function. This second form of the overloaded index operator is only used with `const` objects. It is used for value retrieval instead of value assignment. That, of course, is precisely what we want when using `const` objects. In this situation members are overloaded only by their `const` attribute. This form of function overloading was introduced earlier in the C++ Annotations (sections 2.5.4 and 7.7).

  Since `IntArray` stores values of a primitive type `IntArray::operator[]` const could also have defined a value return type. However, with objects one usually doesn't want the extra copying that's implied with value return types. In those cases `const &` return values are preferred for `const` member functions. So, in the `IntArray` class an `int return value` could have been used as well, resulting in the following prototype:

  ```cpp
  int IntArray::operator[](size_t index) const;
  ```

- As there is only one pointer data member, the destruction of the memory allocated by the object is a simple `delete[]` data.
11.1. OVERLOADING ‘OPERATOR[()]’

Now, the implementation of the members (omitting the trivial implementation of `swap`, cf. chapter 9) are:

```cpp
#include "intarray.ih"

IntArray::IntArray(size_t size)
:    d_size(size)
{    
    if (d_size < 1)
        throw string("IntArray: size of array must be >= 1");

    d_data = new int[d_size];
}

IntArray::IntArray(IntArray const &other)
:    d_size(other.d_size),
    d_data(new int[d_size])
{    
    memcpy(d_data, other.d_data, d_size * sizeof(int));
}

IntArray::~IntArray()
{    
    delete[] d_data;
}

IntArray const &IntArray::operator=(IntArray const &other)
{    
    IntArray tmp(other);
    swap(tmp);
    return *this;
}

int &IntArray::operatorIndex(size_t index) const
{    
    boundary(index);
    return d_data[index];
}

int &IntArray::operator[](size_t index)
{    
    return operatorIndex(index);
}

int const &IntArray::operator[](size_t index) const
{    
    return operatorIndex(index);
}

void IntArray::boundary(size_t index) const
{    
    if (index < d_size)
```
return;
ostringstream out;
out << "IntArray: boundary overflow, index = " <<
    index << ", should be < " << d_size << ' \n';
throw out.str();
}

Note how the operator[] members were implemented: as non-const members may call const member functions and as the implementation of the const member function is identical to the non-const member function’s implementation both operator[] members could be defined inline using an auxiliary function int &operatorIndex(size_t index) const. A const member function may return a non-const reference (or pointer) return value, referring to one of the data members of its object. Of course, this is a potentially dangerous backdoor that may break data hiding. However, the members in the public interface prevent this breach and so the two public operator[] members may themselves safely call the same int &operatorIndex() const member, that defines a private backdoor.

11.2 Overloading the insertion and extraction operators

Classes may be adapted in such a way that their objects may be inserted into and extracted from, respectively, a std::ostream and std::istream.

The class std::ostream defines insertion operators for primitive types, such as int, char *, etc. In this section we learn how to extend the existing functionality of classes (in particular std::istream and std::ostream) in such a way that they can be used also in combination with classes developed much later in history.

In particular we will show how the insertion operator can be overloaded allowing the insertion of any type of object, say Person (see chapter 9), into an ostream. Having defined such an overloaded operator we’re able to use the following code:

Person kr("Kernighan and Ritchie", "unknown", "unknown");

cout << "Name, address and phone number of Person kr:\n" << kr << ' \n';

The statement cout << kr uses operator<<. This member function has two operands: an ostream & and a Person &. The required action is defined in an overloaded free function operator<< expecting two arguments:

// declared in ‘person.h’
std::ostream &operator<<(std::ostream &out, Person const &person);

// defined in some source file
ostream &operator<<(ostream &out, Person const &person)
{
    return
    out <<
        "Name: " << person.name() << ", "
        "Address: " << person.address() << ", "
        "Phone: " << person.phone();
}
11.2. OVERLOADING THE INSERTION AND EXTRACTION OPERATORS

The free function \texttt{operator<<} has the following noteworthy characteristics:

- The function returns a reference to an \texttt{ostream} object, to enable ‘chaining’ of the insertion operator.

- The two operands of \texttt{operator<<} are passed to the free function as its arguments. In the example, the parameter \texttt{out} was initialized by \texttt{cout}, the parameter \texttt{person} by \texttt{kr}.

In order to overload the \texttt{extraction} operator for, e.g., the \texttt{Person} class, members are needed modifying the class’s private data members. Such \textit{modifiers} are normally offered by the class interface. For the \texttt{Person} class these members could be the following:

```cpp
void setName(char const *name);
void setAddress(char const *address);
void setPhone(char const *phone);
```

These members may easily be implemented: the memory pointed to by the corresponding data member must be deleted, and the data member should point to a copy of the text pointed to by the parameter. E.g.,

```cpp
void Person::setAddress(char const *address)
{
    delete[] d_address;
    d_address = strdupnew(address);
}
```

A more elaborate function should check the reasonableness of the new address (\texttt{address} also shouldn’t be a 0-pointer). This however, is not further pursued here. Instead, let’s have a look at the final \texttt{operator>>}. A simple implementation is:

```cpp
istream &operator>>(istream &in, Person &person)
{
    string name;
    string address;
    string phone;

    if (in >> name >> address >> phone) // extract three strings
    {
        person.setName(name.c_str());
        person.setAddress(address.c_str());
        person.setPhone(phone.c_str());
    }
    return in;
}
```

Note the stepwise approach that is followed here. First, the required information is extracted using available extraction operators. Then, if that succeeds, \textit{modifiers} are used to modify the data members of the object to be extracted. Finally, the stream object itself is returned as a reference.
11.3 Conversion operators

A class may be constructed around a built-in type. E.g., a class `String`, constructed around the `char *` type. Such a class may define all kinds of operations, like assignments. Take a look at the following class interface, designed after the `string` class:

```cpp
class String
{
    char *d_string;

public:
    String();
    String(char const *arg);
    ~String();
    String(String const &other);
    String const &operator=(String const &rvalue);
    String const &operator=(char const *rvalue);
};
```

Objects of this class can be initialized from a `char const *`, and also from a `String` itself. There is an overloaded assignment operator, allowing the assignment from a `String` object and from a `char const *`.

Usually, in classes that are less directly linked to their data than this `String` class, there will be an accessor member function, like a member `char const *String::c_str() const`. However, the need to use this latter member doesn’t appeal to our intuition when an array of `String` objects is defined by, e.g., a class `StringArray`. If this latter class provides the `operator[]` to access individual `String` members, it would most likely offer at least the following class interface:

```cpp
class StringArray
{
    String *d_store;
    size_t d_n;

public:
    StringArray(size_t size);
    StringArray(StringArray const &other);
    StringArray const &operator=(StringArray const &rvalue);
    ~StringArray();

    String &operator[](size_t index);
};
```

This interface allows us to assign `String` elements to each other:

```cpp
StringArray sa(10);

sa[4] = sa[3]; // String to String assignment
```

But it is also possible to assign a `char const *` to an element of `sa`:

---

1Note that the assignment from a `char const *` also allows the null-pointer. An assignment like `stringObject = 0` is perfectly in order.
11.3. CONVERSION OPERATORS

\[ sa[3] = "hello world"; \]

Here, the following steps are taken:

- First, \( sa[3] \) is evaluated. This results in a String reference.
- Next, the String class is inspected for an overloaded assignment, expecting a `char const *` to its right-hand side. This operator is found, and the string object \( sa[3] \) receives its new value.

Now we try to do it the other way around: how to access the `char const *` that's stored in \( sa[3] \)? The following attempt fails:

```cpp
char const *cp = sa[3];
```

It fails since we would need an overloaded assignment operator for the 'class `char const *'`. Unfortunately, there isn't such a class, and therefore we can't build that overloaded assignment operator (see also section 11.13). Furthermore, casting won't work as the compiler doesn't know how to cast a String to a char const *. How to proceed?

One possibility is to define an accessor member function `c_str()`:

```cpp
char const *cp = sa[3].c_str();
```

This compiles fine but looks clumsy... A far better approach would be to use a conversion operator.

A conversion operator is a kind of overloaded operator, but this time the overloading is used to cast the object to another type. In class interfaces, the general form of a conversion operator is:

```cpp
operator <type>() const;
```

Conversion operators usually are `const` member functions: they are automatically called when their objects are used as **rvalues** in expressions having a **lvalue**. Using a conversion operator a String object may be interpreted as a `char const * rvalue`, allowing us to perform the above assignment.

Conversion operators are somewhat dangerous. The conversion is automatically performed by the compiler and unless its use is perfectly transparent it may confuse those who read code in which conversion operators are used. E.g., novice C++ programmers are frequently confused by statements like 'if (cin) ...'.

As a rule of thumb: classes should define at most one conversion operator. Multiple conversion operators may be defined but frequently result in ambiguous code. E.g., if a class defines `operator bool()` const and `operator int()` const then passing an object of this class to a function expecting a `size_t` argument results in an ambiguity as an int and a bool may both be used to initialize a size_t.

In the current example, the class String could define the following conversion operator for `char const *`:

```cpp
String::operator char const *() const
{
    return d_string;
}
```
Notes:

- Conversion operators do not define return types. The conversion operator returns a value of the type specified beyond the `operator` keyword.

- In certain situations (e.g., when a `String` argument is passed to a function specifying an ellipsis parameter) the compiler needs a hand to disambiguate our intentions. A `static_cast` solves the problem.

- With `template` functions conversion operators may not work immediately as expected. For example, when defining a conversion operator `X::operator std::string const() const` then `cout << X()` won’t compile. The reason for this is explained in section 21.9, but a shortcut allowing the conversion operator to work is to define the following overloaded `operator<<` function:

```cpp
std::ostream &operator<<(std::ostream &out, std::string const &str) {
    return out.write(str.data(), str.length());
}
```

Conversion operators are also used when objects of classes defining conversion operators are inserted into streams. Realize that the right hand sides of insertion operators are function parameters that are initialized by the operator’s right hand side arguments. The rules are simple:

- If a class `X` defining a conversion operator also defines an insertion operator accepting an `X` object the insertion operator is used;

- Otherwise, if the type returned by the conversion operator is insertable then the conversion operator is used;

- Otherwise, a compilation error results. Note that this happens if the type returned by the conversion operator itself defines a conversion operator to a type that may be inserted into a stream.

In the following example an object of class `Insertable` is directly inserted; an object of the class `Convertor` uses the conversion operator; an object of the class `Error` cannot be inserted since it does not define an insertion operator and the type returned by its conversion operator cannot be inserted either (`Text` does define an `operator int() const`, but the fact that a `Text` itself cannot be inserted causes the error):

```cpp
#include <iostream>
#include <string>
using namespace std;

struct Insertable
{
    operator int() const
    {
        cout << "op int()\n";
    }
};
ostream &operator<<(ostream &out, Insertable const &ins) {
    return out << "insertion operator";
}
```
11.4. THE KEYWORD ‘EXPLICIT’

Some final remarks regarding conversion operators:

- A conversion operator should be a ‘natural extension’ of the facilities of the object. For example, the stream classes define operator bool(), allowing constructions like if (cin).

- A conversion operator should return an *rvalue*. It should do so to enforce data-hiding and because it is the intended use of the conversion operator. Defining a conversion operator as an *lvalue* (e.g., defining an operator int &() conversion operator) opens up a back door, and the operator can only be used as *lvalue* when explicitly called (as in: x.operator int&() = 5). Don’t use it.

- Conversion operators should be defined as *const* member functions as they don’t modify their object’s data members.

- Conversion operators returning composed objects should return const references to these objects whenever possible to avoid calling the composed object’s copy constructor.

11.4 The keyword ‘explicit’

Conversions are not only performed by conversion operators, but also by constructors accepting one argument (i.e., constructors having one or multiple parameters, specifying default argument values
for all parameters or for all but the first parameter).

Assume a database class `DataBase` is defined in which `Person` objects can be stored. It defines a `Person *d_data` pointer, and so it offers a copy constructor and an overloaded assignment operator.

In addition to the copy constructor `DataBase` offers a default constructor and several additional constructors:

- `DataBase(Person const &): the `DataBase` initially contains a single `Person` object;
- `DataBase(istream &in): the data about multiple persons are read from `in`.
- `DataBase(size_t count, istream &in = cin): the data of `count` persons are read from `in`, by default the standard input stream.

The above constructors all are perfectly reasonable. But they also allow the compiler to compile the following code without producing any warning at all:

```
DataBase db;
DataBase db2;
Person person;

db2 = db;  // 1
db2 = person;  // 2
db2 = 10;  // 3
db2 = cin;  // 4
```

Statement 1 is perfectly reasonable: `db` is used to redefine `db2`. Statement 2 might be understandable since we designed `DataBase` to contain `Person` objects. Nevertheless, we might question the logic that's used here as a `Person` is not some kind of `DataBase`. The logic becomes even more opaque when looking at statements 3 and 4. Statement 3 in effect waits for the data of 10 persons to appear at the standard input stream. Nothing like that is suggested by `db2 = 10`.

All four statements are the result of implicit promotions. Since constructors accepting, respectively a `Person`, an `istream`, and a `size_t` and an `istream` have been defined for `DataBase` and since the assignment operator expects a `DataBase` right-hand side (rhs) argument the compiler first converts the rhs arguments to anonymous `DataBase` objects which are then assigned to `db2`.

It is good practice to prevent implicit promotions by using the `explicit` modifier when declaring a constructor. Constructors using the `explicit` modifier can only be used to construct objects explicitly. Statements 2-4 would not have compiled if the constructors expecting one argument would have been declared using `explicit`. E.g.,

```
explicit DataBase(Person const &person);
explicit DataBase(size_t count, std::istream &in);
```

Having declared all constructors accepting one argument as `explicit` the above assignments would have required the explicit specification of the appropriate constructors, thus clarifying the programmer's intent:

```
DataBase db;
DataBase db2;
Person person;
```
11.5. OVERLOADING THE INCREMENT AND DECREMENT OPERATORS

```
db2 = db; // 1
db2 = DataBase(person); // 2
db2 = DataBase(10); // 3
db2 = DataBase(cin); // 4
```

As a rule of thumb prefix one argument constructors with the `explicit` keyword unless implicit promotions are perfectly natural (string’s `char const *` accepting constructor is a case in point).

### 11.4.1 Explicit conversion operators

In addition to explicit constructors, the C++11 standard adds `explicit conversion operators` to C++.

For example, a class might define `operator bool() const` returning `true` if an object of that class is in a usable state and `false` if not. Since the type `bool` is an arithmetic type this could result in unexpected or unintended behavior. Consider:

```cpp
class StreamHandler
{
public:
    operator bool() const; // true: object is fit for use
    ...
};

int fun(StreamHandler &sh)
{
    int sx;
    if (sh) // intended use of operator bool()
        ... use sh as usual; also use ‘sx’
        process(sh); // typo: ‘sx’ was intended
}
```

In this example `process` unintentionally receives the value returned by `operator bool()` using the implicit conversion from `bool` to `int`.

With explicit conversion operators implicit conversions like the one shown in the example are prevented and such conversion operators can only be used in situations where the converted type is explicitly required. E.g., in the condition sections of `if` or repetition statements where a `bool` value is expected. In such cases an `explicit operator bool()` conversion operator would automatically be used.

### 11.5 Overloading the increment and decrement operators

Overloading the increment operator (`operator++`) and decrement operator (`operator--`) introduces a small problem: there are two versions of each operator, as they may be used as `postfix operator` (e.g., `x++`) or as `prefix operator` (e.g., `++x`).

Used as `postfix` operator, the value’s object is returned as an `rvalue`, temporary const object and the post-incremented variable itself disappears from view. Used as `prefix` operator, the variable is incremented, and its value is returned as `lvalue` and it may be altered again by modifying the
prefix operator’s return value. Whereas these characteristics are not required when the operator is overloaded, it is strongly advised to implement these characteristics in any overloaded increment or decrement operator.

Suppose we define a wrapper class around the size_t value type. Such a class could offer the following (partially shown) interface:

```cpp
class Unsigned
{
    size_t d_value;

    public:
    Unsigned();
    explicit Unsigned(size_t init);
    Unsigned &operator++();
};
```

The class’s last member declares the prefix overloaded increment operator. The returned lvalue is Unsigned &. The member is easily implemented:

```cpp
Unsigned &Unsigned::operator++()
{
    ++d_value;
    return *this;
}
```

To define the postfix operator, an overloaded version of the operator is defined, expecting a (dummy) int argument. This might be considered a kludge, or an acceptable application of function overloading. Whatever your opinion in this matter, the following can be concluded:

- Overloaded increment and decrement operators without parameters are prefix operators, and should return references to the current object.
- Overloaded increment and decrement operators having an int parameter are postfix operators, and should return a value which is a copy of the object at the point where its postfix operator is used.

The postfix increment operator is declared as follows in the class Unsigned’s interface:

```cpp
Unsigned operator++(int);
```

It may be implemented as follows:

```cpp
Unsigned Unsigned::operator++(int)
{
    Unsigned tmp(*this);
    ++d_value;
    return tmp;
}
```

Note that the operator’s parameter is not used. It is only part of the implementation to disambiguate the prefix- and postfix operators in implementations and declarations.
In the above example the statement incrementing the current object offers the *nothrow* guarantee as it only involves an operation on a primitive type. If the initial copy construction throws then the original object is not modified, if the return statement throws the object has safely been modified. But incrementing an object could itself throw exceptions. How to implement the increment operators in that case? Once again, *swap* is our friend. Here are the pre- and postfix operators offering the strong guarantee when the member *increment* performing the increment operation may throw:

```cpp
Unsigned &Unsigned::operator++()
{
    Unsigned tmp(*this);
    tmp.increment();
    swap(tmp);
    return *this;
}
Unsigned Unsigned::operator++(int)
{
    Unsigned tmp(*this);
    tmp.increment();
    swap(tmp);
    return tmp;
}
```

The postfix increment operator first creates a copy of the current object. That copy is incremented and then swapped with the current object. If *increment* throws the current object remains unaltered; the swap operation ensures that the original object is returned and the current object becomes the incremented object.

When calling the increment or decrement operator using its full member function name then any *int* argument passed to the function results in calling the postfix operator. Omitting the argument results in calling the prefix operator. Example:

```cpp
Unsigned uns(13);
uns.operator++();   // prefix-incrementing uns
uns.operator++(0); // postfix-incrementing uns
```

### 11.6 Overloading binary operators

In various classes overloading binary operators (like *operator+*) can be a very natural extension of the class's functionality. For example, the *std::string* class has various overloaded forms of *operator+*.

Most binary operators come in two flavors: the plain binary operator (like the + operator) and the compound assignment variant (like the += operator). Whereas the plain binary operators return values, the compound assignment operators return a reference to the object to which the operator was applied. For example, with *std::string* objects the following code (annotations below the example) may be used:

```cpp
std::string s1;
std::string s2;
std::string s3;
```
CHAPTER 11. MORE OPERATOR OVERLOADING

\[
\begin{align*}
s1 &= s2 += s3; & // 1 \\
(s2 += s3) + " \text{postfix}"; & // 2 \\
s1 &= "\text{prefix } + s3; & // 3 \\
"\text{prefix } + s3 + \text{"postfix"}; & // 4
\end{align*}
\]

- at // 1 the contents of \(s3\) is added to \(s2\). Next, \(s2\) is returned, and its new contents are assigned to \(s1\). Note that \(+=\) returns \(s2\) itself.
- at // 2 the contents of \(s3\) is also added to \(s2\), but as \(+=\) returns \(s2\) itself, it's possible to add some more to \(s2\)
- at // 3 the \(+\) operator returns a \texttt{std::string} containing the concatenation of the text \texttt{prefix} and the contents of \(s3\). This string returned by the \(+\) operator is thereupon assigned to \(s1\).
- at // 4 the \(+\) operator is applied twice. The effect is:
  1. The first \(+\) returns a \texttt{std::string} containing the concatenation of the text \texttt{prefix} and the contents of \(s3\).
  2. The second \(+\) operator takes this returned string as its left hand value, and returns a string containing the concatenated text of its left and right hand operands.
  3. The string returned by the second \(+\) operator represents the value of the expression.

Consider the following code, in which a class \texttt{Binary} supports an overloaded \texttt{operator+}:

```cpp
class Binary
{
    public:
        Binary();
        Binary(int value);
        Binary operator+(Binary const &rvalue);
    
    int main()
    {
        Binary b1;
        Binary b2(5);

        b1 = b2 + 3;  // 1
        b1 = 3 + b2;  // 2
    }
}  
```

Compilation of this little program fails for statement // 2, with the compiler reporting an error like:

```
error: no match for ‘operator+’ in ‘3 + b2’
```

Why is statement // 1 compiled correctly whereas statement // 2 won't compile?

In order to understand this remember \textit{promotions}. As we have seen in section 11.4, constructors expecting a single argument may be implicitly activated when an argument of an appropriate type is provided. We've encountered this repeatedly with \texttt{std::string} objects, where an NTBS may be used to initialize a \texttt{std::string} object.
Analogously, in statement // 1, the + operator is called for the b2 object. This operator expects another Binary object as its right hand operand. However, an int is provided. As a constructor Binary(int) exists, the int value is first promoted to a Binary object. Next, this Binary object is passed as argument to the operator+ member.

In statement // 2 no promotions are available: here the + operator is applied to an lvalue that is an int. An int is a primitive type and primitive types have no concept of 'constructors', 'member functions' or 'promotions'.

How, then, are promotions of left-hand operands implemented in statements like "prefix " + s3? Since promotions are applied to function arguments, we must make sure that both operands of binary operators are arguments. This implies that plain binary operators supporting promotions for either their left-hand side operand or right-hand side operand should be declared as free operators, also called free functions.

Functions like the plain binary operators conceptually belong to the class for which they implement the binary operator. Consequently they should be declared in the class's header file. We cover their implementations shortly, but here is our first revision of the declaration of the class Binary, declaring an overloaded + operator as a free function:

```cpp
class Binary
{
  public:
    Binary();
    Binary(int value);
  
  Binary operator+(Binary const &lhs, Binary const &rhs);
};
```

By defining binary operators as free functions, the following promotions are possible:

- If the left-hand operand is of the intended class type, the right hand argument is promoted whenever possible;
- If the right-hand operand is of the intended class type, the left hand argument is promoted whenever possible;
- No promotions occur when none of the operands are of the intended class type;
- An ambiguity occurs when promotions to different classes are possible for the two operands. For example:

```cpp
class A;

class B
{
  public:
    B(A const &a);
};
class A
{
  public:
    A();
    A(B const &b);
};
```
A operator+(A const &a, B const &b);
B operator+(B const &b, A const &a);

int main()
{
    A a;
    a + a;
}

Here, both overloaded + operators are possible when compiling the statement \( a + a \). The ambiguity must be solved by explicitly promoting one of the arguments, e.g., \( a + B(a) \) allows the compiler to resolve the ambiguity to the first overloaded + operator.

The next step is to implement the corresponding overloaded binary assignment operator, having the form \@=, with \@ being a binary operator. As this operator always has a left-hand operand which is an object of its own class, it is implemented as a true member function. Furthermore, the compound assignment operator should return a reference to the object to which the binary operation applies, as the object might be modified in the same statement. E.g., \((s2 += s3) + " postfix"\). Here is our second revision of the class Binary, showing both the declaration of the plain binary operator and the corresponding compound assignment operator:

```cpp
class Binary
{
    public:
        Binary();
        Binary(int value);
        Binary &operator+=(Binary const &rhs);
    }

    Binary operator+(Binary const &lhs, Binary const &rhs);
}
```

How should the compound addition assignment operator be implemented? When implementing the compound assignment operator the strong guarantee should again be kept in mind. Use a temporary object and swap if the add member might throw. Example:

```cpp
Binary &operator+=(Binary const &other)
{
    Binary tmp(*this);
    tmp.add(other);  // this may throw
    swap(tmp);
    return *this;
}
```

It's easy to implement the plain binary operator for classes offering the matching compound assignment operator: the lhs argument is copied into a Binary tmp to which the rhs operand is added. Then tmp is returned. The copy construction and two statements could be contracted into one single return statement, but then compilers usually aren't able to apply copy elision in this case. But copy elision is usually used when the steps are taken separately:

```cpp
class Binary
{...}
```
11.6. OVERLOADING BINARY OPERATORS

```cpp
{  
    public:
        Binary();
        Binary(int value);

        Binary &operator+=(Binary const &other);
    
};

Binary operator+(Binary const &lhs, Binary const &rhs)
{  
    Binary tmp(lhs);
    tmp += rhs;
    return tmp;
}
```

But wait! Remember the design principle for move-aware classes that was given in section 9.7.8? When implementing binary operators we're doing exactly that what was mentioned in that design principle. A temporary object is constructed and the compound assignment operation is applied to the temporary object.

If the class `Binary` is a move-aware class then we can add a move-aware binary operator to our class at very little cost. The actual work is performed by the compound addition assignment operator, as described. However, this operator is called from the move-aware binary operator having prototype

```cpp
Binary operator+(Binary &&lhs, Binary const &rhs);
```

The traditional binary operator's implementation now simply consists of two steps:

- A copy of the left-hand side operand is made using the class's copy constructor;
- The move-aware binary operator is called, passing it the anonymized copy as its left-hand side operand and returning its result as the binary operator's result.

Here is the declaration and implementation of the traditional and move-aware compound addition assignment operator of the class `Binary` for `operator+`:

```cpp
class Binary
{
    public:
        Binary();
        Binary(int value);
            // copy and move constructors are available by default, or
            // they can be explicitly declared and implemented.
        Binary &operator+=(Binary const &other);  // see the text
    
};

inline Binary operator+(Binary &&lhs, Binary const &rhs)
{  
    return std::move(lhs += rhs);  // avoids copy construction
}

inline Binary operator+(Binary const &lhs, Binary const &rhs)
```
11.7 Overloading ‘operator new(size_t)’

When `operator new` is overloaded, it must define a `void *` return type, and its first parameter must be of type `size_t`. The default `operator new` defines only one parameter, but overloaded versions may define multiple parameters. The first one is not explicitly specified but is deducted from the size of objects of the class for which `operator new` is overloaded. In this section overloading `operator new` is discussed. Overloading `new[]` is discussed in section 11.9.

It is possible to define multiple versions of the `operator new`, as long as each version defines its own unique set of arguments. When overloaded `operator new` members must dynamically allocate memory they can do so using the global `operator new`, applying the scope resolution operator `::`

In the next example the overloaded `operator new` of the class `String` initializes the substrate of dynamically allocated `String` objects to 0-bytes:

```cpp
#include <cstddef>
#include <iosfwd>

class String {
    std::string *d_data;

public:
    void *operator new(size_t size) {
        return memset(::operator new(size), 0, size);
    }
    bool empty() const {
        return d_data == 0;
    }
};
```

The above `operator new` is used in the following program, illustrating that even though `String`’s default constructor does nothing the object’s data are initialized to zeroes:

```cpp
#include "string.h"
#include <iostream>
using namespace std;

int main() {
    String *sp = new String;
    cout << boolalpha << sp->empty() << 'n'; // shows: true
}
```

At `new String` the following took place:
• First, String::operator new was called, allocating and initializing a block of memory, the size of a String object.

• Next, a pointer to this block of memory was passed to the (default) String constructor. Since no constructor was defined, the constructor itself didn’t do anything at all.

As String::operator new initialized the allocated memory to zero bytes the allocated String object’s d_data member had already been initialized to a 0-pointer by the time it started to exist.

All member functions (including constructors and destructors) we’ve encountered so far define a (hidden) pointer to the object on which they should operate. This hidden pointer becomes the function’s this pointer.

In the next example of *pseudo* C++ code, the pointer is explicitly shown to illustrate what’s happening when operator new is used. In the first part a String object str is directly defined, in the second part of the example the (overloaded) operator new is used:

```cpp
String::String(String *const this); // real prototype of the default // constructor

String *sp = new String; // This statement is implemented // as follows:

String *sp = static_cast<String*>( // allocation
    String::operator new(sizeof(String))
); // initialization
```

In the above fragment the member functions were treated as object-less member functions of the class String. Such members are called static member functions (cf. chapter 8). Actually, operator new is such a static member function. Since it has no this pointer it cannot reach data members of the object for which it is expected to make memory available. It can only allocate and initialize the allocated memory, but cannot reach the object’s data members by name as there is as yet no data object layout defined.

Following the allocation, the memory is passed (as the this pointer) to the constructor for further processing.

Operator new can have multiple parameters. The first parameter is initialized as an implicit argument and is always a size_t parameter. Additional overloaded operators may define additional parameters. An interesting additional operator new is the placement new operator. With the placement new operator a block of memory has already been set aside and one of the class's constructors is used to initialize that memory. Overloading placement new requires an operator new having two parameters: size_t and char *, pointing to the memory that was already available. The size_t parameter is implicitly initialized, but the remaining parameters must explicitly be initialized using arguments to operator new. Hence we reach the familiar syntactical form of the placement new operator in use:

```cpp
char buffer[sizeof(String)]; // predefined memory
String *sp = new(buffer) String; // placement new call
```

The declaration of the placement new operator in our class String looks like this:

```cpp
void *operator new(size_t size, char *memory);```
It could be implemented like this (also initializing the String's memory to 0-bytes):

```cpp
void *String::operator new(size_t size, char *memory)
{
    return memset(memory, 0, size);
}
```

Any other overloaded version of `operator new` could also be defined. Here is an example showing the use and definition of an overloaded `operator new` storing the object's address immediately in an existing array of pointers to String objects (assuming the array is large enough):

```cpp
// use:
String *next(String **pointers, size_t *idx)
{
    return new(pointers, (*idx)++) String;
}

// implementation:
void *String::operator new(size_t size, String **pointers, size_t idx)
{
    return pointers[idx] = ::operator new(size);
}
```

### 11.8 Overloading ‘operator delete(void *)’

The `delete` operator may also be overloaded. In fact, it's good practice to overload `operator delete` whenever `operator new` is also overloaded.

Operator `delete` must define a `void *` parameter. A second overloaded version defining a second parameter of type `size_t` is related to overloading `operator new[]` and is discussed in section 11.9.

Overloaded `operator delete` members return `void`.

The 'home-made' `operator delete` is called when deleting a dynamically allocated object after executing the destructor of the associated class. So, the statement

```cpp
delete ptr;
```

with `ptr` being a pointer to an object of the class `String` for which the `operator delete` was overloaded, is a shorthand for the following statements:

```cpp
ptr->~String(); // call the class’s destructor

// and do things with the memory pointed to by ptr
String::operator delete(ptr);
```

The overloaded `operator delete` may do whatever it wants to do with the memory pointed to by `ptr`. It could, e.g., simply delete it. If that would be the preferred thing to do, then the default `delete` operator can be called using the `::` scope resolution operator. For example:

```cpp
void String::operator delete(void *ptr)
```
11.9. OPERATORS ‘NEW[]’ AND ‘DELETE[]’

To declare the above overloaded operator delete simply add the following line to the class's interface:

```c
void operator delete(void *ptr);
```

Like operator new operator delete is a static member function (see also chapter 8).

11.9 Operators ‘new[]’ and ‘delete[]’

In sections 9.1.1, 9.1.2 and 9.2.1 operator new[] and operator delete[] were introduced. Like operator new and operator delete the operators new[] and delete[] may be overloaded.

As it is possible to overload new[] and delete[] as well as operator new and operator delete, one should be careful in selecting the appropriate set of operators. The following rule of thumb should always be applied:

If new is used to allocate memory, delete should be used to deallocate memory. If new[] is used to allocate memory, delete[] should be used to deallocate memory.

By default these operators act as follows:

- **operator new** is used to allocate a single object or primitive value. With an object, the object’s constructor is called.
- **operator delete** is used to return the memory allocated by operator new. Again, with class-type objects, the class’s destructor is called.
- **operator new[]** is used to allocate a series of primitive values or objects. If a series of objects is allocated, the class’s default constructor is called to initialize each object individually.
- **operator delete[]** is used to delete the memory previously allocated by new[]. If objects were previously allocated, then the destructor is called for each individual object. Be careful, though, when pointers to objects were allocated. If pointers to objects were allocated the destructors of the objects to which the allocated pointers point won’t automatically be called. A pointer is a primitive type and so no further action is taken when it is returned to the common pool.

11.9.1 Overloading ‘new[]’

To overload operator new[] in a class (e.g., in the class String) add the following line to the class's interface:

```c
void *operator new[](size_t size);
```
The member’s size parameter is implicitly provided and is initialized by C++’s run-time system to the amount of memory that must be allocated. Like the simple one-object operator new it should return a void *. The number of objects that must be initialized can easily be computed from size / sizeof(String) (and of course replacing String by the appropriate class name when overloading operator new[] for another class). The overloaded new[] member may allocate raw memory using e.g., the default operator new[] or the default operator new:

```c++
void *operator new[](size_t size)
{
    return ::operator new[](size);
    // alternatively:
    // return ::operator new(size);
}
```

Before returning the allocated memory the overloaded operator new[] has a chance to do something special. It could, e.g., initialize the memory to zero-bytes.

Once the overloaded operator new[] has been defined, it is automatically used in statements like:

```c++
String *op = new String[12];
```

Like operator new additional overloads of operator new[] may be defined. One opportunity for an operator new[] overload is overloading placement new specifically for arrays of objects. This operator is available by default but becomes unavailable once at least one overloaded operator new[] is defined. Implementing placement new is not difficult. Here is an example, initializing the available memory to 0-bytes before returning:

```c++
void *String::operator new[](size_t size, char *memory)
{
    return memset(memory, 0, size);
}
```

To use this overloaded operator, the second parameter must again be provided, as in:

```c++
char buffer[12 * sizeof(String)];
String *sp = new(buffer) String[12];
```

### 11.9.2 Overloading ‘delete[]’

To overload operator delete[] in a class String add the following line to the class’s interface:

```c++
void operator delete[](void *memory);
```

Its parameter is initialized to the address of a block of memory previously allocated by String::new[].

There are some subtleties to be aware of when implementing operator delete[]. Although the addresses returned by new and new[] point to the allocated object(s), there is an additional size_t value available immediately before the address returned by new and new[]. This size_t value is part of the allocated block and contains the actual size of the block. This of course does not hold true for the placement new operator.
When a class defines a destructor the `size_t` value preceding the address returned by `new[]` does not contain the size of the allocated block, but the number of objects specified when calling `new[]`. Normally that is of no interest, but when overloading `operator delete[]` it might become a useful piece of information. In those cases `operator delete[]` does not receive the address returned by `new[]` but rather the address of the initial `size_t` value. Whether this is at all useful is not clear. By the time `delete[]`'s code is executed all objects have already been destroyed, so `operator delete[]` is only to determine how many objects were destroyed but the objects themselves cannot be used anymore.

Here is an example showing this behavior of `operator delete[]` for a minimal `Demo` class:

```cpp
struct Demo
{
    size_t idx;
    Demo()
    {
        cout << "default cons\n";
    }
    ~Demo()
    {
        cout << "destructor\n";
    }
    void *operator new[](size_t size)
    {
        return ::operator new(size);
    }
    void operator delete[](void *vp)
    {
        cout << "delete[] for: " << vp << '\n';
        ::operator delete[](vp);
    }
};

int main()
{
    Demo *xp;
    cout << ((int *) (xp = new Demo[3]))[-1] << '\n' << '\n';
    cout << xp << '\n';
    cout << "================\n";
    delete[] xp;
}
```

// This program displays (your 0x?????? addresses might differ, but
// the difference between the two should be sizeof(size_t)):
// default cons
// default cons
// default cons
// 3
// 0x8bdd00c
// ===============
// destructor
// destructor
// destructor
// delete[] for: 0x8bdd008
Having overloaded `operator delete[]` for a class `String`, it will be used automatically in statements like:

```cpp
delete[] new String[5];
```

Operator `delete[]` may also be overloaded using an additional `size_t` parameter:

```cpp
void operator delete[](void *p, size_t size);
```

Here `size` is automatically initialized to the size (in bytes) of the block of memory to which `void *p` points. If this form is defined, then `void operator[](void *)` should **not** be defined, to avoid ambiguities. An example of this latter form of `operator delete[]` is:

```cpp
void String::operator delete[](void *p, size_t size)
{
    cout << "deleting " << size << " bytes\n";
    ::operator delete[](ptr);
}
```

Additional overloads of `operator delete[]` may be defined, but to use them they must explicitly be called as static member functions (cf. chapter 8). Example:

```cpp
// declaration:
void String::operator delete[](void *p, ostream &out);
// usage:
String *xp = new String[3];
String::operator delete[](xp, cout);
```

### 11.9.3 ‘new[]’, ‘delete[]’ and exceptions

When an exception is thrown while executing a `new[]` expression, what will happen? In this section we’ll show that `new[]` is exception safe even when only some of the objects were properly constructed.

To begin, `new[]` might throw while trying to allocate the required memory. In this case a `bad_alloc` is thrown and we don’t leak as nothing was allocated.

Having allocated the required memory the class’s default constructor is going to be used for each of the objects in turn. At some point a constructor might throw. What happens next is defined by the C++ standard: the destructors of the already constructed objects are called and the memory allocated for the objects themselves is returned to the common pool. Assuming that the failing constructor offers the basic guarantee `new[]` is therefore exception safe even if a constructor may throw.

The following example illustrates this behavior. A request to allocate and initialize five objects is made, but after constructing two objects construction fails by throwing an exception. The output shows that the destructors of properly constructed objects are called and that the allocated **substrate memory** is properly returned:

```cpp
#include <iostream>
using namespace std;
```
11.9. OPERATORS 'NEW[]' AND 'DELETE[]'

static size_t count = 0;

class X
{
    int x;

public:
    X()
    {
        if (count == 2)
            throw 1;
        cout << "Object " << ++count << '\n';
    }

    ~X()
    {
        cout << "Destroyed " << this << '\n';
    }

    void *operator new[](size_t size)
    {
        cout << "Allocating objects: " << size << " bytes\n";
        return ::operator new(size);
    }

    void operator delete[](void *mem)
    {
        cout << "Deleting memory at " << mem << ", containing: " <<
            *static_cast<int *>(mem) << '\n';
        ::operator delete(mem);
    }
};

int main()
try
{
    X *xp = new X[5];
    cout << "Memory at " << xp << '\n';
    delete[] xp;
}
catch (...){
    cout << "Caught exception.\n";
}

// Output from this program (your 0x??? addresses might differ)
// Allocating objects: 24 bytes
// Object 1
// Object 2
// Destroyed 0x8428010
// Destroyed 0x842800c
// Deleting memory at 0x8428008, containing: 5
// Caught exception.
Function Objects are created by overloading the function call operator `operator()`. By defining the function call operator an object masquerades as a function, hence the term `function objects`. Function objects are also known as `functors`.

Function objects are important when using `generic algorithms`. The use of function objects is preferred over alternatives like pointers to functions. The fact that they are important in the context of generic algorithms leaves us with a didactic dilemma. At this point in the C++ Annotations it would have been nice if generic algorithms would already have been covered, but for the discussion of the generic algorithms knowledge of function objects is required. This bootstrapping problem is solved in a well-known way: by ignoring the dependency for the time being, for now concentrating on the function object concept.

Function objects are objects for which `operator()` has been defined. Function objects are not just used in combination with generic algorithms, but also as a (preferred) alternative to pointers to functions.

Function objects are frequently used to implement `predicate` functions. Predicate functions return boolean values. Predicate functions and predicate function objects are commonly referred to as 'predicates'. Predicates are frequently used by generic algorithms such as the `count_if` generic algorithm, covered in chapter 19, returning the number of times its function object has returned `true`. In the `standard template library` two kinds of predicates are used: `unary predicates` receive one argument, `binary predicates` receive two arguments.

Assume we have a class `Person` and an array of `Person` objects. Further assume that the array is not sorted. A well-known procedure for finding a particular `Person` object in the array is to use the function `lsearch`, which performs a `linear search` in an array. Example:

```cpp
Person &target = targetPerson(); // determine the person to find
Person *pArray;
size_t n = fillPerson(&pArray);

cout << "The target person is";

if (!lsearch(&target, pArray, &n, sizeof(Person), compareFunction))
    cout << " not";
cout << "found\n";
```

The function `targetPerson` determines the person we're looking for, and `fillPerson` is called to fill the array. Then `lsearch` is used to locate the target person.

The comparison function must be available, as its address is one of the arguments of `lsearch`. It must be a real function having an address. If it is defined inline then the compiler has no choice but to ignore that request as inline functions don't have addresses. `CompareFunction` could be implemented like this:

```cpp
int compareFunction(void const *p1, void const *p2)
{
    return *static_cast<Person const *>(p1) != *static_cast<Person const *>(p2);
}
```

This, of course, assumes that the `operator!=` has been overloaded in the class `Person`. But over-
loading \texttt{operator!=} is no big deal, so let's assume that that operator is actually available.

On average $n / 2$ times at least the following actions take place:

1. The two arguments of the compare function are pushed on the stack;
2. The value of the final parameter of \texttt{lsearch} is determined, producing \texttt{compareFunction}'s address;
3. The compare function is called;
4. Then, inside the compare function the address of the right-hand argument of the \texttt{Person::operator!=} argument is pushed on the stack;
5. \texttt{Person::operator!=} is evaluated;
6. The argument of the \texttt{Person::operator!=} function is popped off the stack;
7. The two arguments of the compare function are popped off the stack.

Using function objects results in a different picture. Assume we have constructed a function \texttt{PersonSearch}, having the following prototype (this, however, is not the preferred approach. Normally a generic algorithm is preferred over a home-made function. But for now we focus on \texttt{PersonSearch} to illustrate the use and implementation of a function object):

\begin{verbatim}
Person const *PersonSearch(Person *base, size_t nmemb,
                           Person const &target);
\end{verbatim}

This function can be used as follows:

\begin{verbatim}
Person &target = targetPerson();
Person *pArray;
size_t n = fillPerson(&pArray);
cout << "The target person is";
if (!PersonSearch(pArray, n, target))
  cout << " not";
cout << "found
";
\end{verbatim}

So far, not much has been changed. We've replaced the call to \texttt{lsearch} with a call to another function: \texttt{PersonSearch}. Now look at \texttt{PersonSearch} itself:

\begin{verbatim}
Person const *PersonSearch(Person *base, size_t nmemb,
                           Person const &target)
{
    for (int idx = 0; idx < nmemb; ++idx)
        if (target(base[idx]))
            return base + idx;
    return 0;
}
\end{verbatim}
PersonSearch implements a plain linear search. However, in the for-loop we see `target(base[idx])`. Here `target` is used as a *function object*. Its implementation is simple:

```cpp
bool Person::operator()(Person const &other) const
{
    return *this == other;
}
```

Note the somewhat peculiar syntax: `operator()`. The first set of parentheses define the operator that is overloaded: the function call operator. The second set of parentheses define the parameters that are required for this overloaded operator. In the class header file this overloaded operator is declared as:

```cpp
bool operator()(Person const &other) const;
```

Clearly `Person::operator()` is a simple function. It contains but one statement, and we could consider defining it inline. Assuming we do, then this is what happens when `operator()` is called:

1. The address of the right-hand argument of the `Person::operator==` argument is pushed on the stack;
2. The `operator==` function is evaluated (which probably also is a *semantic* improvement over calling `operator!=` when looking for an object *equal to* a specified target object);
3. The argument of `Person::operator==` argument is popped off the stack.

Due to the fact that `operator()` is an inline function, it is not actually called. Instead `operator==` is called immediately. Moreover, the required stack operations are fairly modest.

Function objects may truly be defined inline. Functions that are called indirectly (i.e., using pointers to functions) can never be defined inline as their addresses must be known. Therefore, even if the function object needs to do very little work it is defined as an ordinary function if it is going to be called through pointers. The overhead of performing the indirect call may annihilate the advantage of the flexibility of calling functions indirectly. In these cases using inline function objects can result in an increase of a program’s efficiency.

An added benefit of function objects is that they may access the private data of their objects. In a search algorithm where a compare function is used (as with `lssearch`) the target and array elements are passed to the compare function using pointers, involving extra stack handling. Using function objects, the target person doesn’t vary within a single search task. Therefore, the target person could be passed to the function object’s class constructor. This is in fact what happens in the expression `target(base[idx])` receiving as its only argument the subsequent elements of the array to search.

### 11.10.1 Constructing manipulators

In chapter 6 we saw constructions like `cout << hex << 13 <<` to display the value 13 in hexadecimal format. One may wonder by what magic the `hex` manipulator accomplishes this. In this section the construction of manipulators like `hex` is covered.

Actually the construction of a manipulator is rather simple. To start, a definition of the manipulator is needed. Let’s assume we want to create a manipulator `w10` which sets the field width of the next field to be written by the `ostream` object to 10. This manipulator is constructed as a function. The `w10` function needs to know about the `ostream` object in which the width must be set. By providing
the function with an `ostream &` parameter, it obtains this knowledge. Now that the function knows about the `ostream` object we’re referring to, it can set the width in that object.

Next, it must be possible to use the manipulator in an insertion sequence. This implies that the return value of the manipulator must be a reference to an `ostream` object also.

From the above considerations we’re now able to construct our `w10` function:

```cpp
#include <ostream>
#include <iomanip>

std::ostream &w10(std::ostream &str)
{
    return str << std::setw(10);
}
```

The `w10` function can of course be used in a ‘stand alone’ mode, but it can also be used as a manipulator. E.g.,

```cpp
#include <iostream>
#include <iomanip>
using namespace std;
extern ostream &w10(ostream &str);

int main()
{
    w10(cout) << 3 << " ships sailed to America\n";
    cout << "And " << w10 << 3 << " more ships sailed too.\n";
}
```

The `w10` function can be used as a manipulator because the `class ostream` has an overloaded `operator<<` accepting a pointer to a function expecting an `ostream &` and returning an `ostream &`. Its definition is:

```cpp
ostream& operator<<(ostream &(*func)(ostream &str))
{
    return (*func)(*this);
}
```

In addition to the above overloaded `operator<<` another one is defined

```cpp
ios_base &operator<<(ios_base &(*func)(ios_base &base))
{
    (*func)(*this);
    return *this;
}
```

This latter function is used when inserting, e.g., `hex` or `internal`.

The above procedure does not work for manipulators requiring arguments. It is of course possible to overload `operator<<` to accept an `ostream` reference and the address of a function expecting an
CHAPTER 11. MORE OPERATOR OVERLOADING

ostream & and, e.g., an int, but while the address of such a function may be specified with the \( << \)-operator, the arguments itself cannot be specified. So, one wonders how the following construction has been implemented:

\[
\text{cout} \ensuremath{\text{<< setprecision}}(3)
\]

In this case the manipulator is defined as a macro. Macro’s, however, are the realm of the preprocessor, and may easily suffer from unwelcome side-effects. In C++ programs they should be avoided whenever possible. The following section introduces a way to implement manipulators requiring arguments without resorting to macros, but using anonymous objects.

11.10.1.1 Manipulators requiring arguments

Manipulators taking arguments are implemented as macros: they are handled by the preprocessor, and are not available beyond the preprocessing stage. The problem appears to be that you can’t call a function in an insertion sequence: when using multiple \( \text{operator} \triangleleft \)-operators in one statement the compiler calls the functions, saves their return values, and then uses their return values in the insertion sequence. That invalidates the ordering of the arguments passed to your \( \text{operator} \triangleleft \)-operators.

So, one might consider constructing another overloaded \( \text{operator} \triangleleft \) accepting the address of a function receiving not just the ostream reference, but a series of other arguments as well. But this creates the problem that it isn’t clear how the function should receive its arguments: you can’t just call it since that takes us back to the above-mentioned problem. Merely passing its address is fine, but then no arguments can be passed to the function.

There exists a solution, based on the use of anonymous objects:

- First, a class is constructed, e.g. Align, whose constructor expects multiple arguments. In our example representing, respectively, the field width and the alignment.
- Furthermore, we define the function:

\[
\text{ostream & operator<<(ostream & ostr, Align const & align)}
\]

so we can insert an Align object into the ostream.

Here is an example of a little program using such a home-made manipulator expecting multiple arguments:

```cpp
#include <iostream>
#include <iomanip>

class Align
{
    unsigned d_width;
    std::ios::fmtflags d_alignment;

public:
    Align(unsigned width, std::ios::fmtflags alignment);
    std::ostream & operator()(std::ostream & ostr) const;
};

Align::Align(unsigned width, std::ios::fmtflags alignment)
```
11.11  THE CASE OF [IO]FSTREAM::OPEN()

:  
  d_width(width),
  d_alignment(alignment)
{}

std::ostream &Align::operator()(std::ostream &ostr) const
{
  ostr.setf(d_alignment, std::ios::adjustfield);
  return ostr << std::setw(d_width);
}

std::ostream &operator<<(std::ostream &ostr, Align const &align)
{
  return align(ostr);
}

using namespace std;

int main()
{
  cout
    << "" << Align(5, ios::left) << "hi" << ""
    << "" << Align(10, ios::right) << "there" << "\n";
  
  /*
   * Generated output:
   * 'hi ' ' there'
  */
}

Note that in order to insert an anonymous Align object into the ostream, the operator<< function must define a Align const & parameter (note the const modifier).

11.11  The case of [IO]fstream::open()

Earlier, in section 6.4.2.1, it was noted that the [IO]fstream::open members expect an ios::openmode value as their final argument. E.g., to open an fstream object for writing you could do as follows:

```cpp
fstream out;
out.open("/tmp/out", ios::out);
```

Combinations are also possible. To open an fstream object for both reading and writing the following stanza is often seen:

```cpp
fstream out;
out.open("/tmp/out", ios::in | ios::out);
```

When trying to combine enum values using a 'home made' enum we may run into problems. Consider the following:
enum Permission
{
    READ = 1 << 0,
    WRITE = 1 << 1,
    EXECUTE = 1 << 2
};

void setPermission(Permission permission);

int main()
{
    setPermission(READ | WRITE);
}

When offering this little program to the compiler it replies with an error message like this:

    invalid conversion from 'int' to 'Permission'

The question is of course: why is it OK to combine ios::openmode values passing these combined values to the stream’s open member, but not OK to combine Permission values.

Combining enum values using arithmetic operators results in int-typed values. Conceptually this never was our intention. Conceptually it can be considered correct to combine enum values if the resulting value conceptually makes sense as a value that is still within the original enumeration domain. Note that after adding a value READWRITE = READ | WRITE to the above enum we’re still not allowed to specify READ | WRITE as an argument to setPermission.

To answer the question about combining enumeration values and yet stay within the enumeration’s domain we turn to operator overloading. Up to this point operator overloading has been applied to class types. Free functions like operator« have been overloaded, and those overloads are conceptually within the domain of their class.

As C++ is a strongly typed language realize that defining an enum is really something beyond the mere association of int-values with symbolic names. An enumeration type is really a type of its own, and as with any type its operators can be overloaded. When writing READ | WRITE the compiler performs the default conversion from enum values to int values and applies the operator to ints. It does so when it has no alternative.

But it is also possible to overload the enum type’s operators. Thus we may ensure that we’ll remain within the enum’s domain even though the resulting value wasn’t defined by the enum. The advantage of type-safety and conceptual clarity is considered to outweigh the somewhat peculiar introduction of values hitherto not defined by the enum.

Here is an example of such an overloaded operator:

    Permission operator|(Permission left, Permission right)
    {
        return static_cast<Permission>(static_cast<int>(left) | right);
    }

Other operators can easily and analogously be constructed.

Operators like the above were defined for the ios::openmode enumeration type, allowing us to specify ios::in | ios::out as argument to open while specifying the corresponding parameter as ios::openmode as well. Clearly, operator overloading can be used in many situations, not necessarily only involving class-types.
11.12. USER-DEFINED LITERALS

289

11.12 User-defined literals
The C++11 standard offers user-defined literals, also known as extensible literals. Standard C++
defines various kinds of literals, like numerical constants (with or without suffixes), character constants and string (textual) literals.
A user-defined literal is defined by a function (see also section 23.3) that must be defined at namespace scope. Such a function is called a literal operator. A literal operator cannot be a class member
function. Under the C++11 standard the names of a literal operator must start with an underscore,
and a literal operator is used (called) by suffixing its name (including the underscore) to the argument that must be passed to it . Assuming _NM2km (nautical mile to km) is the name of a literal
operator, then it could be called as 100_NM2km, producing, e.g., the value 185.2.
Using Type to represent the return type of the literal operator its generic declaration looks like this:
Type operator "" _identifier(parameter-list);
The blank space trailing the empty string is required. The parameter lists of literal operators can
be:
• unsigned long long int. It is used as, e.g., 123_identifier. The argument to this literal
operator can be decimal constants, binary constants (initial 0b), octal constants (initial 0) and
hexadecimal constants (initial 0x);
• long double. It is used as, e.g., 12.25_NM2km;
• char const *text. The text argument is an NTBS. It is used as, e.g., 1234_pental. The
argument must not be given double quotes, and must represent a numeric constant, as also
expected by literal operators defining unsigned long long int parameters.
• char const *text, size_t len. Here, the compiler determines len as if it had called
strlen(text). It is used as, e.g., "hello"_nVowels;
• wchar_t const *text, size_t len, same as the previous one, but accepting a string of
wchar_t characters. It is used as, e.g., L"1234"_charSum;
• char16_t const *text, size_t len, same as the previous one, but accepting a string of
char16_t characters. It is used as, e.g., u"utf 16"_uc;
• char32_t const *text, size_t len, same as the previous one, but accepting a string of
char32_t characters. It is used as, e.g., U"UTF 32"_lc;
If literal operators are overloaded the compiler will pick the literal operator requiring the least
‘effort’. E.g., 120 is processed by a literal operator defining a unsigned long long int parameter
and not by its overloaded version, defining a char const * parameter. But if overloaded literal
operators exist defining char const * and long double parameters then the operator defining a
char const * parameter is used when the argument 120 is provided, while the operator defining
a long double parameter is used with the argument 120.3.
A literator operator can define any return type. Here is an example of a definition of the _NM2km
literal operator:
double operator "" _NM2km(char const *nm)
{
return std::stod(nm) * 1.852;


290  CHAPTER 11. MORE OPERATOR OVERLOADING

Of course, the argument could also have been a long double constant. Here’s an alternative implementation, explicitly expecting a long double:

```cpp
double constexpr operator "" _NM2km(long double nm)
{
    return nm * 1.852;
}
```

double value = 450.5_NM2km;  // example of use

A numeric constant can also be processed completely at compile-time. Section 23.3 provides the details of this type of literal operator.

Arguments to literal operators are themselves always constants. A literal operator like _NM2km cannot be used to convert, e.g., the value of a variable. A literal operator, although it is defined as function, cannot be called like a function. The following examples therefore result in compilation errors:

```cpp
double speed;

speed_NM2km; // no identifier ‘speed_NM2km’
_NM2km(speed); // no function _NM2km
_NM2km(120.3); // no function _NM2km
```

### 11.13 Overloadable operators

The following operators can be overloaded:

| +    | -    | *    | /    | %    | ^    | &    | | |
|------|------|------|------|------|------|------|---|
| ~    | !    | ,    | =    | <    | >    | <=   | >= |
| ++   | --   | <<=  | >>=  | %=   | ^=   | &=   | |= |
| <<=  | >>=  | []   | ()   | ->   | -*   | new   | new[] |

dele te delete[]

<table>
<thead>
<tr>
<th>textual alternative</th>
<th>operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>and_eq</td>
<td>&amp;=</td>
</tr>
<tr>
<td>bitand</td>
<td>&amp;</td>
</tr>
<tr>
<td>bitor</td>
<td></td>
</tr>
<tr>
<td>compl</td>
<td>~</td>
</tr>
<tr>
<td>not</td>
<td>!</td>
</tr>
<tr>
<td>not_eq</td>
<td>!=</td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
<tr>
<td>or_eq</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>^</td>
</tr>
<tr>
<td>xor_eq</td>
<td>^=</td>
</tr>
</tbody>
</table>

Several operators have textual alternatives: “Textual” alternatives of operators are also overloadable (e.g., operator and). However, note that textual alternatives
are not additional operators. So, within the same context operator& and operator and can not both be overloaded.

Several of these operators may only be overloaded as member functions within a class. This holds true for the ‘=’, the ‘[]’, the ‘()’ and the ‘->’ operators. Consequently, it isn’t possible to redefine, e.g., the assignment operator globally in such a way that it accepts a char const * as an lvalue and a String & as an rvalue. Fortunately, that isn’t necessary either, as we have seen in section 11.3.

Finally, the following operators cannot be overloaded:

.  .*  ::  ?:  sizeof  typeid
Chapter 12

Abstract Containers

C++ offers several predefined datatypes, all part of the Standard Template Library, which can be used to implement solutions to frequently occurring problems. The datatypes discussed in this chapter are all containers: you can put stuff inside them, and you can retrieve the stored information from them.

The interesting part is that the kind of data that can be stored inside these containers has been left unspecified at the time the containers were constructed. That’s why they are spoken of as abstract containers.

Abstract containers rely heavily on templates, covered in chapter 21 and beyond. To use abstract containers, only a minimal grasp of the template concept is required. In C++ a template is in fact a recipe for constructing a function or a complete class. The recipe tries to abstract the functionality of the class or function as much as possible from the data on which the class or function operates. As the data types on which the templates operate were not known when the template was implemented, the datatypes are either inferred from the context in which a function template is used, or they are mentioned explicitly when a class template is used (the term that's used here is instantiated). In situations where the types are explicitly mentioned, the angle bracket notation is used to indicate which data types are required. For example, below (in section 12.2) we’ll encounter the pair container, which requires the explicit mentioning of two data types. Here is a pair object containing both an int and a string:

```cpp
pair<int, string> myPair;
```

The object myPair is defined as an object holding both an int and a string.

The angle bracket notation is used intensively in the upcoming discussion of abstract containers. Actually, understanding this part of templates is the only real requirement for using abstract containers. Now that we’ve introduced this notation, we can postpone the more thorough discussion of templates to chapter 21, and concentrate on their use in this chapter.

Most of the abstract containers are sequential containers: they contain data that can be stored and retrieved in some sequential way. Examples are the array, implementing a fixed-sized array; a vector, implementing an extendable array; the list, implementing a data structure that allows for the easy insertion or deletion of data; the queue, also called a FIFO (first in, first out) structure, in which the first element that is entered is the first element to be retrieved again; and the stack, which is a first in, last out (FILO or LIFO) structure.

In addition to sequential containers several special containers are available. The pair is a basic container in which a pair of values (of types that are left open for further specification) can be
stored, like two strings, two ints, a string and a double, etc. Pairs are often used to return data elements that naturally come in pairs. For example, the map is an abstract container storing keys and their associated values. Elements of these maps are returned as pairs.

A variant of the pair is the complex container, implementing operations that are defined on complex numbers.

A tuple (cf. section 22.6) generalizes the pair container to a data structure accommodating any number of different data types.

All abstract containers described in this chapter as well as the string and stream datatypes (cf. chapters 5 and 6) are part of the Standard Template Library.

All but the unordered containers support the following basic set of operators:

- The overloaded assignment operator, so we can assign two containers of the same types to each other. If the container's data type supports move assignment, then assignment of an anonymous temporary container to a destination container will use move assignment when assigning new values to the destination container's element. Overloaded assignment is also supported by the unordered containers;

- Tests for equality: == and != The equality operator applied to two containers returns true if the two containers have the same number of elements, which are pairwise equal according to the equality operator of the contained data type. The inequality operator does the opposite;

- Ordering operators: <, <=, > and >=. The < operator returns true if each element in the left-hand side container is less than each corresponding element in the right-hand side container. Additional elements in either the left-hand side container or the right-hand side container are ignored.

```cpp
container left;
container right;

left = {0, 2, 4};
right = {1, 3}; // left < right

right = {1, 3, 6, 1, 2}; // left < right
```

Note that before a user-defined type (usually a class-type) can be stored in a container, the user-defined type should at least support:

- A default value (e.g., a default constructor)
- The equality operator (==)
- The less-than operator (<)

With the advent of the C++11 standard sequential containers can also be initialized using initializer lists.

Most containers (exceptions are the stack (section 12.4.11), priority_queue (section 12.4.5), and queue (section 12.4.4) containers) support members to determine their maximum sizes (through their member function max_size).

Virtually all containers support copy construction. If the container supports copy construction and the container's data type supports move construction, then move construction is automatically used.
for the container’s data elements when a container is initialized with an anonymous temporary container.

Closely linked to the standard template library are the \textit{generic algorithms}. These algorithms may be used to perform frequently occurring tasks or more complex tasks than is possible with the containers themselves, like counting, filling, merging, filtering etc.. An overview of generic algorithms and their applications is given in chapter 19. Generic algorithms usually rely on the availability of \textit{iterators}, representing begin and end-points for processing data stored inside containers. The abstract containers usually support constructors and members expecting iterators, and they often have members returning iterators (comparable to the \texttt{string::begin} and \texttt{string::end} members). In this chapter the iterator concept is not further investigated. Refer to chapter 18 for this.

The url \url{http://www.sgi.com/Technology/STL} is worth visiting as it offers more extensive coverage of abstract containers and the standard template library than can be provided by the C++ annotations.

Containers often collect data during their lifetimes. When a container goes out of scope, its destructor tries to destroy its data elements. This only succeeds if the data elements themselves are stored inside the container. If the data elements of containers are pointers to dynamically allocated memory then the memory pointed to by these pointers is not destroyed, resulting in a memory leak. A consequence of this scheme is that the data stored in a container should often be considered the ‘property’ of the container: the container should be able to destroy its data elements when the container’s destructor is called. So, normally containers should not contain pointers to data. Also, a container should not be required to contain \texttt{const} data, as \texttt{const} data prevent the use of many of the container’s members, like the assignment operator.

\section*{Notations used in this chapter}

In this chapter about containers, the following notational conventions are used:

- Containers live in the standard namespace. In code examples this will be clearly visible, but in the text \texttt{std::} is usually omitted.

- A container without angle brackets represents any container of that type. Mentally add the required type in angle bracket notation. E.g., \texttt{pair} may represent \texttt{pair<string, int>}.

- The notation \texttt{Type} represents the generic type. \texttt{Type} could be \texttt{int}, \texttt{string}, etc.

- Identifiers \texttt{object} and \texttt{container} represent objects of the container type under discussion.

- The identifier \texttt{value} represents a value of the type that is stored in the container.

- Simple, one-letter identifiers, like \texttt{n} represent unsigned values.

- Longer identifiers represent iterator iterators. Examples are \texttt{pos}, \texttt{from}, \texttt{beyond}

Some containers, e.g., the \texttt{map} container, contain pairs of values, usually called ‘keys’ and ‘values’. For such containers the following notational convention is used in addition:

- The identifier \texttt{key} indicates a value of the used key-type

- The identifier \texttt{keyvalue} indicates a value of the ‘\texttt{value_type}’ used with the particular container.
12.2 The ‘pair’ container

The pair container is a rather basic container. It is used to store two elements, called first and second, and that’s about it. Before using pair containers the header file <utility> must be included.

The pair’s data types are specified when the pair object is defined (or declared) using the template’s angle bracket notation (cf. chapter 21). Examples:

```
pair<string, string> piper("PA28", "PH-ANI");
pair<string, string> cessna("C172", "PH-ANG");
```

describe the variables piper and cessna as pair variables containing two strings. Both strings can be retrieved using the first and second fields of the pair type:

```
cout << piper.first << \n << // shows ‘PA28’
cessna.second << \n; // shows ‘PH-ANG’
```

The first and second members can also be used to reassign values:

```
cessna.first = "C152";
cessna.second = "PH-ANW";
```

If a pair object must be completely reassigned, an anonymous pair object can be used as the right-hand operand of the assignment. An anonymous variable defines a temporary variable (which receives no name) solely for the purpose of (re)assigning another variable of the same type. Its generic form is

```
type(initializer list)
```

Note that when a pair object is used the type specification is not completed by just mentioning the container name pair. It also requires the specification of the data types which are stored within the pair. For this the (template) angle bracket notation is used again. E.g., the reassignment of the cessna pair variable could have been accomplished as follows:

```
cessna = pair<string, string>("C152", "PH-ANW");
```

In cases like these, the type specification can become quite elaborate, which has caused a revival of interest in the possibilities offered by the typedef keyword. If many pair<type1, type2> clauses are used in a source, the typing effort may be reduced and readability might be improved by first defining a name for the clause, and then using the defined name later. E.g.,

```
typedef pair<string, string> pairStrStr;
cessna = pairStrStr("C152", "PH-ANW");
```

Apart from this (and the basic set of operations (assignment and comparisons)) the pair offers no further functionality. It is, however, a basic ingredient of the upcoming abstract containers map, multimap and hash_map.

The C++11 standard offers a generalized pair container: the tuple, covered in section 22.6.
12.3 Allocators

Most containers use a special object for allocating the memory that is managed by them. This object is called an allocator, and its type is (usually by default) specified when a container is constructed. A container's allocator can be obtained using the container's `get_allocator` member, which returns a copy of the allocator used by the container. Allocators offer the following members:

- `value_type *address(value_type &object)`
  
  returns the address of `object`.

- `value_type *allocate(size_t count)`
  
  allocates raw memory for holding `count` values of the container's `value_type`.

- `void construct(value_type *object, Arg &&...args)`
  
  using placement new, uses the arguments following `object` to install a value at `object`.

- `void destroy(value_type *object)`
  
  calls `object`'s destructor (but doesn't deallocate `object`'s own memory).

- `void deallocate(value_type *object, size_t count)`
  
  calls `operator delete` to delete `object`'s memory, previously allocated by `allocate`.

- `size_t max_size()`
  
  returns the maximum number of elements that `allocate` can allocate.

Here is an example, using the allocator of a vector of strings (see section 12.4.2 below for a description of the `vector` container):

```cpp
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<string> vs;
    auto allocator = vs.get_allocator(); // get the allocator
    string *sp = allocator.allocate(3); // alloc. space for 3 strings
    allocator.construct(&sp[0], "hello world"); // initialize 1st string
    allocator.construct(&sp[1], sp[0]); // use the copy constructor
    allocator.construct(&sp[2], 12, '='); // string of 12 = chars

    cout << sp[0] << "\n" << // show the strings
         sp[1] << "\n" <<
         sp[2] << "\n" <<
```
"could have allocated " << allocator.max_size() << " strings\n"

for (size_t idx = 0; idx != 3; ++idx)
  allocator.destroy(sp + idx); // delete the string's
  // contents

  allocator.deallocate(sp, 3); // and delete sp itself again.
}

12.4 Available Containers

12.4.1 The ‘array’ container

The array class implements a fixed-size array. Before using the array container the <array> header file must be included.

To define a std::array both the data type of its elements and its size must be specified: the data type is given after an opening angular bracket, immediately following the ‘array’ container name. The array’s size is provided after the data type specification. Finally, a closing angular bracket completes the array’s type. Specifications like this are common practice with containers. The combination of array, type and size defines a type. As a result, array<string, 4> defines another type than array<string, 5>, and a function explicitly defining an array<Type, N> parameter will not accept an array<Type, M> argument if N and M are unequal.

The array’s size may be defined as 0 (although such an array probably has little use as it cannot store any element). The elements of an array are stored contiguously. If array<Type, N> arr has been defined, then &arr[n] + m == &arr[n + m, assuming 0 <= n < N and assuming 0 <= n + m < N.

The following constructors, operators, and member functions are available:

- Constructors:
  - The copy and move constructors are available;
  - A array may be constructed with a fixed number N of default elements:
    array<string, N> object;
  - An initial subset of the elements of an array may be initialized using a brace delimited initializer list:
    array<double, 4> dArr = {1.2, 2.4};
    Here dArr is defined as an array of 4 element, with dArr[0] and dArr[1] initialized to, respectively 1.2 and 2.4, and dArr[2] and dArr[3] initialized to 0. A attractive characteristic of arrays (and other containers) is that containers initialize their data elements to the data type’s default value. The data type’s default constructor is used for this initialization. With non-class data types the value 0 is used. So, for an array<double, 4> array we know that all but its explicitly initialized elements are initialized to zero.

- In addition to the standard operators for containers, the array supports the index operator, which can be used to retrieve or reassign individual elements of the array. Note that the elements which are indexed must exist. For example, having defined an empty array a statement like iarr[0] = 18 produces an error, as the array is empty. Note that operator[] does not respect its array bounds. If you want run-time array bound checking, use the array’s at member.
12.4. AVAILABLE CONTAINERS

- The `std::array` class offers the following member functions:
  - `Type &at(size_t idx)`:
    - returns a reference to the array’s element at index position `idx`. If `idx` exceeds
      the array’s size a `std::out_of_range` exception is thrown.
  - `Type &back()`:
    - returns a reference to the last element in the array. It is the responsibility of the
      programmer to use the member only if the array is not empty.
  - `array::iterator begin()`:
    - returns an iterator pointing to the first element in the array, returning `end` if the
      array is empty.
  - `array::const_iterator cbegin()`:
    - returns a `const_iterator` pointing to the first element in the array, returning `cend`
      if the array is empty.
  - `array::const_iterator cend()`:
    - returns a `const_iterator` pointing just beyond the array’s last element.
  - `array::const_reverse_iterator crbegin()`:
    - returns a `const_reverse_iterator` pointing to the last element in the array, return-
      ing `crend` if the array is empty.
  - `array::const_reverse_iterator crend()`:
    - returns a `const_reverse_iterator` pointing just before the array’s first element.
  - `value_type *data()`:
    - returns a pointer to the array’s first data element. With a const array a
      `value_type const *` is returned.
  - `bool empty()`:
    - returns `true` if the array contains no elements.
  - `array::iterator end()`:
    - returns an iterator pointing beyond the last element in the array.
  - `void fill(Type const &item)`:
    - fills all the array’s elements with a copy of `item`
  - `Type &front()`:
    - returns a reference to the first element in the array. It is the responsibility of the
      programmer to use the member only if the array is not empty.
  - `array::reverse_iterator rbegin()`:
    - this member returns an iterator pointing to the last element in the array.
  - `array::reverse_iterator rend()`:
    - returns an iterator pointing before the first element in the array.
  - `constexpr size_t size()`:
    - returns the number of elements the array contains.
  - `void swap(<array<Type, N> &other)`:
    - swaps the contents of the current and other array. The array other’s data type
      and size must be equal to the data type and size of the object calling `swap`.

Using an `array` rather than a standard C style array offers several advantages:

- All its elements are immediately initialized;
• Introspection is possible (e.g., size can be used);
• The array container can be used in the context of templates, there code is developed that operates on data types that become available only after the code itself has been developed;
• Since array supports reverse iterators, it can be immediately be used with generic algorithms performing ‘reversed’ operations (e.g., to perform a descending rather than ascending sort (cf. section 19.1.58))

In general, when looking for a sequential data structure, the array or vector (introduced in the next section) should be your ‘weapon of choice’. Only if these containers demonstrably do not fit the problem at hand you should use another type of container.

12.4.2 The ‘vector’ container

The vector class implements an expandable array. Before using the vector container the <vector> header file must be included.

The following constructors, operators, and member functions are available:

• Constructors:
  – The copy and move constructors are available;
  – A vector may be constructed empty:
    
    ```
    vector<string> object;
    ```
  – A vector may be initialized to a certain number of elements:
    
    ```
    vector<string> object(5, string("Hello")); // initialize to 5 Hello’s,
    vector<string> container(10); // and to 10 empty strings
    vector<string> names = {"george", "frank", "tony", "karel"};
    ```
  – A vector may be initialized using iterators. To initialize a vector with elements 5 until 10 (including the last one) of an existing vector<string> the following construction may be used:
    
    ```
    extern vector<string> container;
    vector<string> object(&container[5], &container[11]);
    ```

    Note here that the last element pointed to by the second iterator (&container[11]) is not stored in object. This is a simple example of the use of iterators, in which the range of values that is used starts at the first value, and includes all elements up to but not including the element to which the second iterator refers. The standard notation for this is [begin, end).

• In addition to the standard operators for containers, the vector supports the index operator, which can be used to retrieve or reassign individual elements of the vector. Note that the elements which are indexed must exist. For example, having defined an empty vector a statement like ivect[0] = 18 produces an error, as the vector is empty. So, the vector is not automatically expanded, and operator[] does not respect its array bounds. In this case the vector should be resized first, or ivect.push_back(18) should be used (see below). If you need run-time array bound checking, use the vector’s at member.

• The vector class offers the following member functions:

  – void assign(...):
    
    assigns new contents to the vector:
12.4. AVAILABLE CONTAINERS

- *assign(iterator begin, iterator end)* assigns the values at the iterator range [begin, end) to the vector;
- *assign(size_type n, value_type const &val)* assigns n copies of val to the vector;
- *assign(initializer_list<value_type> values)* assigns the values in the initializer list to the vector.

- Type &at(size_t idx):
  returns a reference to the vector's element at index position idx. If idx exceeds the vector's size a std::out_of_range exception is thrown.

- Type &back():
  returns a reference to the last element in the vector. It is the responsibility of the programmer to use the member only if the vector is not empty.

- vector::iterator begin():
  returns an iterator pointing to the first element in the vector, returning end if the vector is empty.

- size_t capacity():
  Number of elements for which memory has been allocated. It returns at least the value returned by size

- vector::const_iterator cbegin():
  returns a const_iterator pointing to the first element in the vector, returning cend if the vector is empty.

- vector::const_iterator cend():
  returns a const_iterator pointing just beyond the vector's last element.

- void clear():
  erases all the vector's elements.

- vector::const_reverse_iterator crbegin():
  returns a const_reverse_iterator pointing to the last element in the vector, returning crend if the vector is empty.

- vector::const_reverse_iterator crend():
  returns a const_reverse_iterator pointing just before the vector's first element.

- value_type *data() :
  returns a pointer to the vector's first data element.

- iterator emplace(const_iterator position, Args &&...args):
  a value_type object is constructed from the arguments specified after position, and the newly created element is inserted at position.

- void emplace_back(Args &&...args):
  a value_type object is constructed from the member's arguments, and the newly created element is inserted beyond the vector's last element.

- bool empty():
  returns true if the vector contains no elements.

- vector::iterator end():
  returns an iterator pointing beyond the last element in the vector.

- vector::iterator erase():
  erases a specific range of elements in the vector:
* erase(pos) erases the element pointed to by the iterator pos. The iterator ++pos is returned.
* erase(first, beyond) erases elements indicated by the iterator range [first, beyond), returning beyond.

- Type &front():
  returns a reference to the first element in the vector. It is the responsibility of the programmer to use the member only if the vector is not empty.
- allocator_type get_allocator() const:
  returns a copy of the allocator object used by the vector object.
- ... insert():
  elements may be inserted starting at a certain position. The return value depends on the version of insert() that is called:
  * vector::iterator insert(pos) inserts a default value of type Type at pos, pos is returned.
  * vector::iterator insert(pos, value) inserts value at pos, pos is returned.
  * void insert(pos, first, beyond) inserts the elements in the iterator range [first, beyond).
  * void insert(pos, n, value) inserts n elements having value value at position pos.
- size_t max_size():
  returns the maximum number of elements this vector may contain.
- void pop_back():
  removes the last element from the vector. With an empty vector nothing happens.
- void push_back(value):
  adds value to the end of the vector.
- vector::reverse_iterator rbegin():
  this member returns an iterator pointing to the last element in the vector.
- vector::reverse_iterator rend():
  returns an iterator pointing before the first element in the vector.
- void reserve(size_t request):
  if request is less than or equal to capacity, this call has no effect. Otherwise, it is a request to allocate additional memory. If the call is successful, then capacity returns a value of at least request. Otherwise, capacity is unchanged. In either case, size’s return value won’t change, until a function like resize is called, actually changing the number of accessible elements.
- void resize():
  can be used to alter the number of elements that are currently stored in the vector:
  * resize(n, value) may be used to resize the vector to a size of n. Value is optional. If the vector is expanded and value is not provided, the additional elements are initialized to the default value of the used data type, otherwise value is used to initialize extra elements.
- void shrink_to_fit():
  optionally reduces the amount of memory allocated by a vector to its current size. The implementor is free to ignore or otherwise optimize this request. In order to guarantee a ‘shrink to fit’ operation the
vector<Type>(vectorObject).swap(vectorObject)
12.4. AVAILABLE CONTAINERS

Figure 12.1: A list data-structure

- size_t size():
  returns the number of elements in the vector.
- void swap():
  swaps two vectors using identical data types. Example:

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> v1(7);
    vector<int> v2(10);

    v1.swap(v2);
    cout << v1.size() << " " << v2.size() << '\n';
}
/*
Produced output:
10 7
*/
```

12.4.3 The ‘list’ container

The list container implements a list data structure. Before using a list container the header file <list> must be included.

The organization of a list is shown in figure 12.1. Figure 12.1 shows that a list consists of separate list-elements, connected by pointers. The list can be traversed in two directions: starting at Front the list may be traversed from left to right, until the 0-pointer is reached at the end of the rightmost list-element. The list can also be traversed from right to left: starting at Back, the list is traversed from right to left, until eventually the 0-pointer emanating from the leftmost list-element is reached.

As a subtlety note that the representation given in figure 12.1 is not necessarily used in actual implementations of the list. For example, consider the following little program:

```cpp
int main()
{
    list<int> l;
    cout << "size: " << l.size() << "", first element: " << l.front() << '\n';
}
```
When this program is run it might actually produce the output:

```
size: 0, first element: 0
```

Its front element can even be assigned a value. In this case the implementor has chosen to provide the list with a hidden element. The list actually is a **circular** list, where the hidden element serves as terminating element, replacing the 0-pointers in figure 12.1. As noted, this is a subtlety, which doesn’t affect the conceptual notion of a list as a data structure ending in 0-pointers. Note also that it is well known that various implementations of list-structures are possible (cf. Aho, A.V., Hopcroft J.E. and Ullman, J.D., (1983) *Data Structures and Algorithms* (Addison-Wesley)).

Both lists and vectors are often appropriate data structures in situations where an unknown number of data elements must be stored. However, there are some rules of thumb to follow when selecting the appropriate data structure.

- When most accesses are random, a vector is the preferred data structure. Example: in a program counting character frequencies in a textfile, a `vector<int> frequencies(256)` is the datastructure of choice, as the values of the received characters can be used as indices into the `frequencies` vector.

- The previous example illustrates a second rule of thumb, also favoring the vector: if the number of elements is known in advance (and does not notably change during the lifetime of the program), the vector is also preferred over the list.

- In cases where insertions or deletions prevail and the data structure is large the list is generally preferred.

At present lists aren’t as useful anymore as they used to be (when computers were much slower and more memory-constrained). Except maybe for some rare cases, a vector should be the preferred container; even when implementing algorithms traditionally using lists.

Other considerations related to the choice between lists and vectors should also be given some thought. Although it is true that the vector is able to grow dynamically, the dynamic growth requires data-copying. Clearly, copying a million large data structures takes a considerable amount of time, even on fast computers. On the other hand, inserting a large number of elements in a list doesn’t require us to copy non-involved data. Inserting a new element in a list merely requires us to juggle some pointers. In figure 12.2 this is shown: a new element is inserted between the second and third element, creating a new list of four elements. Removing an element from a list is also fairly easy. Starting again from the situation shown in figure 12.1, figure 12.3 shows what happens if element two is removed from our list. Again: only pointers need to be juggled. In this case it’s even simpler than adding an element: only two pointers need to be rerouted. To summarize the comparison between lists and vectors: it’s probably best to conclude that there is no clear-cut answer to the question what data structure to prefer. There are rules of thumb, which may be adhered to. But if worse comes to worst, a profiler may be required to find out what’s best.

The list container offers the following constructors, operators, and member functions:

- **Constructors:**
  - The copy and move constructors are available;
  - A list may be constructed empty:
    ```
    list<string> object;
    ```
    As with the vector, it is an error to refer to an element of an empty list.
A list may be initialized to a certain number of elements. By default, if the initialization value is not explicitly mentioned, the default value or default constructor for the actual data type is used. For example:

```cpp
list<string> object(5, string("Hello")); // initialize to 5 Hello’s
list<string> container(10); // and to 10 empty strings
```

A list may be initialized using a two iterators. To initialize a list with elements 5 until 10 (including the last one) of a `vector<string>` the following construction may be used:

```cpp
extern vector<string> container;
list<string> object(&container[5], &container[11]);
```

- The list does not offer specialized operators, apart from the standard operators for containers.
- The following member functions are available:
  - `void assign(...):`
    - assigns new contents to the list:
      * `assign(iterator begin, iterator end)` assigns the values at the iterator range `[begin, end)` to the list;
      * `assign(size_type n, value_type const &val)` assigns n copies of val to the list;
  - `Type &back()`:
    - returns a reference to the last element in the list. It is the responsibility of the programmer to use this member only if the list is not empty.
  - `list::iterator begin()`:
    - returns an iterator pointing to the first element in the list, returning end if the list is empty.
- void clear():
  erases all elements from the list.

- bool empty():
  returns true if the list contains no elements.

- list::iterator end():
  returns an iterator pointing beyond the last element in the list.

- list::iterator erase():
  erases a specific range of elements in the list:
  * erase(pos) erases the element pointed to by pos. The iterator ++pos is returned.
  * erase(first, beyond) erases elements indicated by the iterator range [first, beyond). Beyond is returned.

- Type &front():
  returns a reference to the first element in the list. It is the responsibility of the programmer to use this member only if the list is not empty.

- allocator_type get_allocator() const:
  returns a copy of the allocator object used by the list object.

- ... insert():
  inserts elements into the list. The return value depends on the version of insert that is called:
  * list::iterator insert(pos) inserts a default value of type Type at pos, pos is returned.
  * list::iterator insert(pos, value) inserts value at pos, pos is returned.
  * void insert(pos, first, beyond) inserts the elements in the iterator range [first, beyond).
  * void insert(pos, n, value) inserts n elements having value value at position pos.

- size_t max_size():
  returns the maximum number of elements this list may contain.

- void merge(list<Type> other):
  this member function assumes that the current and other lists are sorted (see below, the member sort). Based on that assumption, it inserts the elements of other into the current list in such a way that the modified list remains sorted. If both list are not sorted, the resulting list will be ordered 'as much as possible', given the initial ordering of the elements in the two lists. list<Type>::merge uses Type::operator< to sort the data in the list, which operator must therefore be available. The next example illustrates the use of the merge member: the list 'object' is not sorted, so the resulting list is ordered 'as much as possible'.

```cpp
#include <iostream>
#include <string>
#include <list>
using namespace std;

void showlist(list<string> &target) {
  for
   (list<string>::iterator from = target.begin();
from != target.end();
++from
)
cout << *from << " ";
cout << '\n';
}

int main()
{
    list<string> first;
    list<string> second;

    first.push_back(string("alpha"));
    first.push_back(string("bravo"));
    first.push_back(string("golf"));
    first.push_back(string("quebec"));

    second.push_back(string("oscar"));
    second.push_back(string("mike"));
    second.push_back(string("november"));
    second.push_back(string("zulu"));

    first.merge(second);
    showlist(first);
}

A subtlety is that merge doesn’t alter the list if the list itself is used as argument:
object.merge(object) won’t change the list ‘object’.

- void pop_back():
    removes the last element from the list. With an empty list nothing happens.

- void pop_front():
    removes the first element from the list. With an empty list nothing happens.

- void push_back(value):
    adds value to the end of the list.

- void push_front(value):
    adds value before the first element of the list.

- list::reverse_iterator rbegin():
    returns an iterator pointing to the last element in the list.

- void remove(value):
    removes all occurrences of value from the list. In the following example, the two
    strings ‘Hello’ are removed from the list object:

    #include <iostream>
    #include <list>
    using namespace std;

    int main()
    {
        list<string> object;
object.push_back("Hello");
object.push_back("World");
object.push_back("Hello");
object.push_back("World");

object.remove("Hello");

while (object.size())
{
    cout << object.front() << 'n';
    object.pop_front();
}

*/

Generated output:
World
World
*/

- void remove_if(Predicate pred):
  removes all occurrences from the list for which the predicate function or function
  object pred returns true. For each of the objects stored in the list the predi-
  cate is called as pred(*iter), where iter represents the iterator used inter-
  nally by remove_if. If a function pred is used, its prototype should be bool
  pred(value_type const &object).

list::reverse_iterator rend():
  this member returns an iterator pointing before the first element in the
  list.

void resize():
  alters the number of elements that are currently stored in the list:
  * resize(n, value) may be used to resize the list to a size of n. Value is
  optional. If the list is expanded and value is not provided, the extra elements
  are initialized to the default value of the used data type, otherwise value is
  used to initialize extra elements.

void reverse():
  reverses the order of the elements in the list. The element back becomes
  front and vice versa.

size_t size():
  returns the number of elements in the list.

void sort():
  sorts the list. Once the list has been sorted, An example of its
  use is given at the description of the unique member function below.
  list<Type>::sort uses Type::operator< to sort the data in the list,
  which operator must therefore be available.

void splice(pos, object):
  transfers the contents of object to the current list, starting the inser-
  tion at the iterator position pos of the object using the splice member.
  Following splice, object is empty. For example:

#include <iostream>
#include <string>
#include <list>
using namespace std;
int main()
{
    list<string> object;

    object.push_front(string("Hello"));
    object.push_back(string("World"));

    list<string> argument(object);
    object.splice(++object.begin(), argument);

    cout << "Object contains " << object.size() << " elements, " <<
         "Argument contains " << argument.size() << " elements,\n";

    while (object.size())
    {
        cout << object.front() << '\n';
        object.pop_front();
    }
}

Alternatively, argument may be followed by an iterator of argument, indicating the first element of argument that should be spliced, or by two iterators begin and end defining the iterator-range [begin, end) on argument that should be spliced into object.

void swap():
swaps two lists using identical data types.

void unique():
operating on a sorted list, this member function removes all consecutively identical elements from the list. list<Type>::unique uses Type::operator== to identify identical data elements, which operator must therefore be available. Here's an example removing all multiply occurring words from the list:

```cpp
#include <iostream>
#include <string>
#include <list>
using namespace std;

void showlist(list<string> &target)
{
    for
    {
        list<string>::iterator from = target.begin();
        from != target.end();
        ++from
    }
    cout << +from << " ";
    cout << '\n';
}
```

int main()
{
string
array[] =
{
  "charley",
  "alpha",
  "bravo",
  "alpha"
};

list<string>
target
{
  array, array + sizeof(array)
   / sizeof(string)
};
cout << "Initially we have:\n";
showlist(target);

target.sort();
cout << "After sort() we have:\n";
showlist(target);

target.unique();
cout << "After unique() we have:\n";
showlist(target);
}
/*
Generated output:
Initially we have:
charley alpha bravo alpha
After sort() we have:
alpha alpha bravo charley
After unique() we have:
alpha bravo charley
*/

12.4.4 The ‘queue’ container

The queue class implements a queue data structure. Before using a queue container the header file <queue> must be included.

A queue is depicted in figure 12.4. In figure 12.4 it is shown that a queue has one point (the back) where items can be added to the queue, and one point (the front) where items can be removed (read) from the queue. A queue is therefore also called a FIFO data structure, for first in, first out. It is most often used in situations where events should be handled in the same order as they are generated.

The following constructors, operators, and member functions are available for the queue container:

- Constructors:
  - The copy and move constructors are available;
12.4. AVAILABLE CONTAINERS

Figure 12.4: A queue data-structure

- A queue may be constructed empty:
  
  ```cpp
  queue<string> object;
  ```

  As with the vector, it is an error to refer to an element of an empty queue.

- The queue container only supports the basic container operators.

- The following member functions are available for queues:

  - `Type &back()`:
    
    returns a reference to the last element in the queue. It is the responsibility of the programmer to use the member only if the queue is not empty.

  - `bool empty()`:
    
    returns true if the queue contains no elements.

  - `Type &front()`:
    
    returns a reference to the first element in the queue. It is the responsibility of the programmer to use the member only if the queue is not empty.

  - `void pop()`:
    
    removes the element at the front of the queue. Note that the element is not returned by this member. Nothing happens if the member is called for an empty queue. One might wonder why `pop` returns `void`, instead of a value of type `Type`. One reason is found in the principles of good software design: functions should perform one task. Combining the removal and return of the removed element breaks this principle. Moreover, when this principle is abandoned `pop`’s implementation is always flawed. Consider the prototypical implementation of a `pop` member that is supposed to return the just popped value:

  ```cpp
  Type queue::pop()
  {
    Type ret(front());
    erase_front();
    return ret;
  }
  ```

  The venom, as usual, is in the tail: since `queue` has no control over `Type`’s behavior the final statement (`return ret`) might throw. By that time the queue’s front element has already been removed from the queue and so it is lost. Thus, a `Type` returning `pop` member cannot offer the strong guarantee and consequently `pop` should not return the former `front` element. Because of all this, we must first use `front` and then `pop` to obtain and remove the queue’s front element.

  - `void push(value)`:
    
    this member adds `value` to the back of the queue.

  - `size_t size()`:
    
    returns the number of elements in the queue.
Note that the queue does not support iterators or a subscript operator. The only elements that can be accessed are its front and back element. A queue can be emptied by:

- repeatedly removing its front element;
- assigning an empty queue using the same data type to it;
- having its destructor called.

### 12.4.5 The ‘priority_queue’ container

The `priority_queue` class implements a priority queue data structure. Before using a `priority_queue` container the `<queue>` header file must have been included.

A priority queue is identical to a `queue`, but allows the entry of data elements according to priority rules. A real-life priority queue is found, e.g., at airport check-in terminals. At a terminal the passengers normally stand in line to wait for their turn to check in, but late passengers are usually allowed to jump the queue: they receive a higher priority than other passengers.

The priority queue uses `operator<` of the data type stored in the priority queue to decide about the priority of the data elements. The `smaller` the value, the `lower` the priority. So, the priority queue could be used to sort values while they arrive. A simple example of such a priority queue application is the following program: it reads words from `cin` and writes a sorted list of words to `cout`:

```cpp
#include <iostream>
#include <string>
#include <queue>
using namespace std;

int main()
{
    priority_queue<string> q;
    string word;

    while (cin >> word)
        q.push(word);

    while (q.size())
    {
        cout << q.top() << '\n';
        q.pop();
    }
}
```

Unfortunately, the words are listed in reversed order: because of the underlying `operator<` operator the words appearing later in the ASCII-sequence appear first in the priority queue. A solution to that problem is to define a `wrapper class` around the `string` datatype, reversing string’s `operator<`. Here is the modified program:

```cpp
#include <iostream>
#include <string>
#include <queue>
```
class Text
{
    std::string d_s;

public:
    Text(std::string const &str)
    :
        d_s(str)
    {}
    operator std::string const &() const
    {
        return d_s;
    }
    bool operator<(Text const &right) const
    {
        return d_s > right.d_s;
    }
};

using namespace std;

int main()
{
    priority_queue<Text> q;
    string word;

    while (cin >> word)
        q.push(word);

    while (q.size())
    {
        word = q.top();
        cout << word << '\n';
        q.pop();
    }
}

Other possibilities to achieve the same exist. One would be to store the contents of the priority queue in, e.g., a vector, from which the elements can be read in reversed order.

The following constructors, operators, and member functions are available for the priority_queue container:

- **Constructors:**
  - The copy and move constructors are available;
  - A `priority_queue` may be constructed empty:
    ```
    priority_queue<string> object;
    ```
    As with the `vector`, it is an error to refer to an element of an empty priority queue.

- **The `priority_queue` only supports the basic operators of containers.**

- The following member functions are available for priority queues:
- bool empty():
  returns true if the priority queue contains no elements.
- void pop():
  removes the element at the top of the priority queue. Note that the element is not returned by this member. Nothing happens if this member is called for an empty priority queue. See section 12.4.4 for a discussion about the reason why pop has return type void.
- void push(value):
  inserts value at the appropriate position in the priority queue.
- size_t size():
  returns the number of elements in the priority queue.
- Type &top():
  returns a reference to the first element of the priority queue. It is the responsibility of the programmer to use the member only if the priority queue is not empty.

Note that the priority queue does not support iterators or a subscript operator. The only element that can be accessed is its top element. A priority queue can be emptied by:

- repeatedly removing its top element;
- assigning an empty queue using the same data type to it;
- having its destructor called.

### 12.4.6 The ‘deque’ container

The deque (pronounce: ‘deck’) class implements a doubly ended queue data structure (deque). Before using a deque container the header file `<deque>` must be included.

A deque is comparable to a queue, but it allows for reading and writing at both ends. Actually, the deque data type supports a lot more functionality than the queue, as illustrated by the following overview of available member functions. A deque is a combination of a vector and two queues, operating at both ends of the vector. In situations where random insertions and the addition and/or removal of elements at one or both sides of the vector occurs frequently using a deque should be considered.

The following constructors, operators, and member functions are available for deques:

- Constructors:
  - The copy and move constructors are available;
  - A deque may be constructed empty:
    ```cpp
deque<string> object;
    ```
    As with the vector, it is an error to refer to an element of an empty deque.
  - A deque may be initialized to a certain number of elements. By default, if the initialization value is not explicitly mentioned, the default value or default constructor for the actual data type is used. For example:
    ```cpp
deque<string> object(5, string("Hello")), // initialize to 5 Hello’s
deque<string> container(10); // and to 10 empty strings
```
A deque may be initialized using two iterators. To initialize a deque with elements 5 until 10 (including the last one) of a `vector<string>` the following construction may be used:

```cpp
extern vector<string> container;
deque<string> object(&container[5], &container[11]);
```

- In addition to the standard operators for containers, the deque supports the index operator, which may be used to retrieve or reassign random elements of the deque. Note that the indexed elements must exist.

- The following member functions are available for deques:
  - `void assign(...)`:
    
    assigns new contents to the deque:

    * `assign(iterator begin, iterator end)` assigns the values at the iterator range `[begin, end)` to the deque;
    * `assign(size_type n, value_type const &val)` assigns `n` copies of `val` to the deque;

  - `Type &at(size_t idx)`:
    
    returns a reference to the deque's element at index position `idx`. If `idx` exceeds the deque's size a `std::out_of_range` exception is thrown.

  - `Type &back()`:
    
    returns a reference to the last element in the deque. It is the responsibility of the programmer to use the member only if the deque is not empty.

  - `deque::iterator begin()`:
    
    returns an iterator pointing to the first element in the deque.

  - `deque::const_iterator cbegin()`:
    
    returns a const_iterator pointing to the first element in the deque, returning `cend` if the deque is empty.

  - `deque::const_iterator cend()`:
    
    returns a const_iterator pointing just beyond the deque's last element.

  - `void clear()`:
    
    erases all elements in the deque.

  - `deque::const_reverse_iterator crbegin()`:
    
    returns a const_reverse_iterator pointing to the last element in the deque, returning `crend` if the deque is empty.

  - `deque::const_reverse_iterator crend()`:
    
    returns a const_reverse_iterator pointing just before the deque's first element.

  - `iterator emplace(const_iterator position, Args &&...args)`
    
    a `value_type` object is constructed from the arguments specified after position, and the newly created element is inserted at position.

  - `void emplace_back(Args &&...args)`
    
    a `value_type` object is constructed from the member's arguments, and the newly created element is inserted beyond the deque's last element.

  - `void emplace_front(Args &&...args)`
    
    a `value_type` object is constructed from the member's arguments, and the newly created element is inserted before the deque's first element.
- bool empty():
  returns true if the deque contains no elements.
- deque::iterator end():
  returns an iterator pointing beyond the last element in the deque.
- deque::iterator erase():
  the member can be used to erase a specific range of elements in the deque:
  * erase(pos) erases the element pointed to by pos. The iterator ++pos is returned.
  * erase(first, beyond) erases elements indicated by the iterator range [first, beyond). Beyond is returned.
- Type &front():
  returns a reference to the first element in the deque. It is the responsibility of the programmer to use the member only if the deque is not empty.
- allocator_type get_allocator() const:
  returns a copy of the allocator object used by the deque object.
- ... insert():
  inserts elements starting at a certain position. The return value depends on the version of insert that is called:
  * deque::iterator insert(pos) inserts a default value of type Type at pos, pos is returned.
  * deque::iterator insert(pos, value) inserts value at pos, pos is returned.
  * void insert(pos, first, beyond) inserts the elements in the iterator range [first, beyond).
  * void insert(pos, n, value) inserts n elements having value value starting at iterator position pos.
- size_t max_size():
  returns the maximum number of elements this deque may contain.
- void pop_back():
  removes the last element from the deque. With an empty deque nothing happens.
- void pop_front():
  removes the first element from the deque. With an empty deque nothing happens.
- void push_back(value):
  adds value to the end of the deque.
- void push_front(value):
  adds value before the first element of the deque.
- deque::reverse_iterator rbegin():
  returns an iterator pointing to the last element in the deque.
- deque::reverse_iterator rend():
  this member returns an iterator pointing before the first element in the deque.
- void resize():
  alters the number of elements that are currently stored in the deque:
  * resize(n, value) may be used to resize the deque to a size of n. Value is optional. If the deque is expanded and value is not provided, the additional elements are initialized to the default value of the used data type, otherwise value is used to initialize extra elements.
12.4. AVAILABLE CONTAINERS

- `void shrink_to_fit()`:
  optionally reduces the amount of memory allocated by a deque to its current size. The implementor is free to ignore or otherwise optimize this request. In order to guarantee a 'shrink to fit' operation `deque<Type>(dequeObject).swap(dequeObject)` idiom can be used.

- `size_t size()`:
  returns the number of elements in the deque.

- `void swap(argument)`:
  swaps two deques using identical data types.

12.4.7 The ‘map’ container

The `map` class offers a (sorted) associative array. Before using a `map` container the `<map>` header file must be included.

A map is filled with key/value pairs, which may be of any container-accepted type. Since types are associated with both the key and the value, we must specify two types in the angle bracket notation, comparable to the specification we've seen with the `pair` container (cf. section 12.2). The first type represents the key's type, the second type represents the value's type. For example, a `map` in which the key is a `string` and the value is a `double` can be defined as follows:

```cpp
map<string, double> object;
```

The `key` is used to access its associated information. That information is called the `value`. For example, a phone book uses the names of people as the key, and uses the telephone number and maybe other information (e.g., the zip-code, the address, the profession) as value. Since a map sorts its keys, the key's `operator<` must be defined, and it must be sensible to use it. For example, it is generally a bad idea to use pointers for keys, as sorting pointers is something different than sorting the values pointed at by those pointers.

The two fundamental operations on maps are the storage of Key/Value combinations, and the retrieval of values, given their keys. The index operator using a key as the index, can be used for both. If the index operator is used as `lvalue`, the expression's `rvalue` is inserted into the map. If it is used as `rvalue`, the key's associated value is retrieved. Each key can be stored only once in a map. If the same key is entered again, the new value replaces the formerly stored value, which is lost.

A specific key/value combination can implicitly or explicitly be inserted into a map. If explicit insertion is required, the key/value combination must be constructed first. For this, every map defines a `value_type` which may be used to create values that can be stored in the map. For example, a value for a `map<string, int>` can be constructed as follows:

```cpp
map<string, int>::value_type siValue("Hello", 1);
```

The `value_type` is associated with the `map<string, int>`: the type of the key is `string`, the type of the value is `int`. Anonymous `value_type` objects are also often used. E.g.,

```cpp
map<string, int>::value_type("Hello", 1);
```

Instead of using the line `map<string, int>::value_type(...) over and over again, a typedef is frequently used to reduce typing and to improve readability:

```cpp
typedef map<string, int>::value_type StringIntValue
```
Using this typedef, values for the `map<string, int>` may now be constructed using:

```
StringIntValue("Hello", 1);
```

Alternatively, `pairs` may be used to represent key/value combinations used by maps:

```
pair<string, int>("Hello", 1);
```

### 12.4.7.1 The ‘map’ constructors

The following constructors are available for the `map` container:

- The copy and move constructors are available;
- A `map` may be constructed empty:

```
map<string, int> object;
```

Note that the values stored in maps may be containers themselves. For example, the following defines a `map` in which the value is a `pair`: a container nested under another container:

```
map<string, pair<string, string>> object;
```

Note the use of the two consecutive closing angle brackets. Before the advent of the C++11 standard consecutive closing brackets in container type specifications (and generally: in the context of template type specifications) resulted in a compilation error, as the immediate concatenation of the two closing angle brackets would be interpreted by the compiler as a right shift operator (`operator>>`), which is not what we want here. In compilers supporting the C++11 standard this construction is accepted. Compilers not yet implementing this feature require a separating blank between two consecutive closing angle brackets.

- A `map` may be initialized using two iterators. The iterators may either point to `value_type` values for the map to be constructed, or to plain `pair` objects. If pairs are used, their first element represents the type of the keys, and their second element represents the type of the values. Example:

```
pair<string, int> pa[] =
{
    pair<string, int>("one", 1),
    pair<string, int>("two", 2),
    pair<string, int>("three", 3),
};

map<string, int> object(&pa[0], &pa[3]);
```

In this example, `map<string, int>::value_type` could have been written instead of `pair<string, int>` as well.

If `begin` represents the first iterator that is used to construct a map and if `end` represents the second iterator, `[begin, end)` will be used to initialize the map. Maybe contrary to intuition, the `map` constructor only enters new keys. If the last element of `pa` would have been "one", 3, only two elements would have entered the map: "one", 1 and "two", 2. The value "one", 3 would silently have been ignored.
The `map` receives its own copies of the data to which the iterators point as illustrated by the following example:

```cpp
#include <iostream>
#include <map>
using namespace std;

class MyClass
{
    public:
        MyClass()
        {
            cout << "MyClass constructor\n";
        }
        MyClass(MyClass const &other)
        {
            cout << "MyClass copy constructor\n";
        }
        ~MyClass()
        {
            cout << "MyClass destructor\n";
        }
};

int main()
{
    pair<string, MyClass> pairs[] =
    {
        pair<string, MyClass>("one", MyClass())
    };
    cout << "pairs constructed\n";
    map<string, MyClass> mapsm(&pairs[0], &pairs[1]);
    cout << "mapsm constructed\n";
} /*
 * Generated output:
 * MyClass constructor
 * MyClass copy constructor
 * MyClass destructor
 * pairs constructed
 * MyClass copy constructor
 * MyClass copy constructor
 * MyClass destructor
 * mapsm constructed
 * MyClass destructor
 * MyClass destructor
 */
```

When tracing the output of this program, we see that, first, the constructor of a `MyClass` object is called to initialize the anonymous element of the array `pairs`. This object is then copied into the first element of the array `pairs` by the copy constructor. Next, the original element is not required anymore and is destroyed. At that point the array `pairs` has been constructed. Thereupon, the `map` constructs a temporary `pair` object, which is used to construct the map.
chapter 12. abstract containers

...element. Having constructed the map element, the temporary pair object is destroyed. Eventually, when the program terminates, the pair element stored in the map is destroyed too.

12.4.7.2 The ‘map’ operators

The map supports, in addition to the standard operators for containers, the index operator.

The index operator may be used to retrieve or reassign individual elements of the map. The argument of the index operator is called a key.

If the provided key is not available in the map, a new data element is automatically added to the map using the default value or default constructor to initialize the value part of the new element. This default value is returned if the index operator is used as an rvalue.

When initializing a new or reassigning another element of the map, the type of the right-hand side of the assignment operator must be equal to (or promotable to) the type of the map's value part. E.g., to add or change the value of element "two" in a map, the following statement can be used:

\[ \text{mapsm["two"]} = \text{MyClass();} \]

12.4.7.3 The ‘map’ public members

The following member functions are available for the map container:

- \text{mapped_type &at(key_type const &key):}
  - returns a reference to the map's mapped_type associated with key. If the key is not stored in the map an std::out_of_range exception is thrown.
- map::iterator begin():
  - returns an iterator pointing to the first element of the map.
- map::const_iterator cbegin():
  - returns a const_iterator pointing to the first element in the map, returning cend if the map is empty.
- map::const_iterator cend():
  - returns a const_iterator pointing just beyond the map's last element.
- \text{void clear():}
  - erases all elements from the map.
- size_t count(key):
  - returns 1 if the provided key is available in the map, otherwise 0 is returned.
- map::reverse_iterator crbegin() const:
  - returns an iterator pointing to the last element of the map.
- map::reverse_iterator crend():
  - returns an iterator pointing before the first element of the map.
12.4. AVAILABLE CONTAINERS

• pair<iterator, bool> emplace(Args &&...args):
  a value_type object is constructed from emplace’s arguments. If the map already
  contained an object using the same key_type value, then a std::pair is returned
  containing an iterator pointing to the object using the same key_type value and the
  value false. If no such key_type value was found, the newly constructed object is
  inserted into the map, and the returned std::pair contains an iterator pointing to
  the newly inserted inserted value_type as well as the value true.

• iterator emplace_hint(const_iterator position, Args &&...args):
  a value_type object is constructed from the member’s arguments, and the newly
  created element is inserted into the map, unless the (at args) provided key already
  exists. The implementation may or may not use position as a hint to start looking
  for an insertion point. The returned iterator points to the value_type using the
  provided key. It may refer to an already existing value_type or to a newly added
  value_type; an existing value_type is not replaced. If a new value was added,
  then the container’s size has been incremented when emplace_hint returns.

• bool empty():
  returns true if the map contains no elements.

• map::iterator end():
  returns an iterator pointing beyond the last element of the map.

• pair<map::iterator, map::iterator> equal_range(key):
  this member returns a pair of iterators, being respectively the return values of the
  member functions lower_bound and upper_bound, introduced below. An example
  illustrating these member functions is given at the discussion of the member function
  upper_bound.

• ... erase():
  erases a specific element or range of elements from the map:
  – bool erase(key) erases the element having the given key from the map. True is re-
    turned if the value was removed, false if the map did not contain an element using the
    given key.
  – void erase(pos) erases the element pointed to by the iterator pos.
  – void erase(first, beyond) erases all elements indicated by the iterator range
    [first, beyond).

• map::iterator find(key):
  returns an iterator to the element having the given key. If the element isn’t avail-
  able, end is returned. The following example illustrates the use of the find member
  function:

```
#include <iostream>
#include <map>
using namespace std;

int main()
{
    map<string, int> object;
```
object["one"] = 1;
map<string, int>::iterator it = object.find("one");
cout << "'one' " <<
    (it == object.end() ? "not " : "") << "found\n";
it = object.find("three");
cout << "'three' " <<
    (it == object.end() ? "not " : "") << "found\n";
} /*
 * Generated output:
 * 'one' found
 * 'three' not found
 */

• allocator_type get_allocator() const:

    returns a copy of the allocator object used by the map object.

• ... insert():

    inserts elements into the map. Values associated with already
    existing keys, however, are not replaced by new values. Its return value
    depends on the version of insert that is called:

- pair<map::iterator, bool> insert(keyvalue) inserts a new value_type into
  the map. The return value is a pair<map::iterator, bool>. If the returned
  bool field is true, keyvalue was inserted into the map. The value false
  indicates that the key that was specified in keyvalue was already available
  in the map, and so keyvalue was not inserted into the map. In both cases the
  map::iterator field points to the data
  element having the key that was specified in keyvalue. The use of this
  variant of insert is illustrated by the following example:

#include <iostream>
#include <string>
#include <map>
using namespace std;

int main()
{
    pair<string, int> pa[] =
    {
        pair<string,int>("one", 10),
        pair<string,int>("two", 20),
        pair<string,int>("three", 30),
    };
    map<string, int> object(&pa[0], &pa[3]);

    // (four, 40) and 'true' is returned
    pair<map<string, int>::iterator, bool>
    ret = object.insert
    {
12.4. AVAILABLE CONTAINERS

map<string, int>::value_type
("four", 40)
);

cout << boolalpha;

cout << ret.first->first << " " <<
   ret.first->second << " " <<
   ret.second << " " << object["four"] << '\n';

// {four, 40} and 'false' is returned
ret = object.insert
{
    map<string, int>::value_type
    ("four", 0)
};

cout << ret.first->first << " " <<
   ret.first->second << " " <<
   ret.second << " " << object["four"] << '\n';
/*

Generated output:
four 40 true 40
four 40 false 40
*/

Note the somewhat peculiar constructions like

cout << ret.first->first << " " << ret.first->second << ...

Note that 'ret' is equal to the pair returned by the insert member function. Its 'first' field is an iterator into the map<string, int>, so it can be considered a pointer to a map<string, int>::value_type. These value types themselves are pairs too, having 'first' and 'second' fields. Consequently, 'ret.first->first' is the key of the map value (a string), and 'ret.first->second' is the value (an int).

- map::iterator insert(pos, keyvalue). This way a map::value_type may also be inserted into the map. pos is ignored, and an iterator to the inserted element is returned.

- void insert(first, beyond) inserts the (map::value_type) elements pointed to by the iterator range [first, beyond).

• key_compare key_comp():

returns a copy of the object used by the map to compare keys. The type map<KeyType, ValueType>::key_compare is defined by the map container and key_compare's parameters have types KeyType const &. The comparison function returns true if the first key argument should be ordered before the second key argument. To compare keys and values, use value_comp, listed below.

• map::iterator lower_bound(key):

returns an iterator pointing to the first keyvalue element of which the key is at least equal to the specified key. If no such element exists, the function returns end.
• size_t max_size():
  returns the maximum number of elements this map may contain.

• map::reverse_iterator rbegin():
  returns an iterator pointing to the last element of the map.

• map::reverse_iterator rend():
  returns an iterator pointing before the first element of the map.

• size_t size():
  returns the number of elements in the map.

• void swap(argument):
  swaps two maps using identical key/value types.

• map::iterator upper_bound(key):
  returns an iterator pointing to the first key value element having a key exceeding the specified key. If no such element exists, the function returns end. The following example illustrates the member functions equal_range, lower_bound and upper_bound:

```cpp
#include <iostream>
#include <map>
using namespace std;

int main()
{
  pair<string, int> pa[] =
  {
    pair<string, int>("one", 10),
    pair<string, int>("two", 20),
    pair<string, int>("three", 30),
  };
  map<string, int> object(&pa[0], &pa[3]);
  map<string, int>::iterator it;
  if ((it = object.lower_bound("tw")) != object.end())
    cout << "lower-bound 'tw' is available, it is: " << it->first << '
';
  if (object.lower_bound("twoo") == object.end())
    cout << "lower-bound 'twoo' not available" << '
';
  cout << "lower-bound two: " "
    object.lower_bound("two")->first " " is available\n";
  if ((it = object.upper_bound("tw")) != object.end())
    cout << "upper-bound 'tw' is available, it is: " << it->first << '
';
  if (object.upper_bound("twoo") == object.end())
    cout << "upper-bound 'twoo' not available" << '
';
```
if (object.upper_bound("two") == object.end())
    cout << "upper-bound 'two' not available" << '
';

pair
<
    map<string, int>::iterator,
    map<string, int>::iterator
>
p = object.equal_range("two");

Generated output:
lower-bound 'tw' is available, it is: two
lower-bound 'twooo' not available
lower-bound two: two is available
upper-bound 'tw' is available, it is: two
upper-bound 'twooo' not available
upper-bound 'two' not available
equal range: 'first' points to two, 'second' is not available

• value_compare value_comp():

returns a copy of the object used by the map to compare keys. The type
map<KeyType, ValueType>::value_compare is defined by the map container
and value_compare's parameters have types value_type const &. The compar-
ison function returns true if the first key argument should be ordered before the
second key argument. The Value_Type elements of the value_type objects passed
to this member are not used by the returned function.

12.4.7.4 The 'map': a simple example

As mentioned at the beginning of section 12.4.7, the map represents a sorted associative array. In
a map the keys are sorted. If an application must visit all elements in a map the begin and end
iterators must be used.

The following example illustrates how to make a simple table listing all keys and values found in a
map:

#include <iostream>
#include <iomanip>
#include <map>
using namespace std;

int main()
{
    pair<string, int> pa[] =
    {
        pair<string,int>("one", 10),
        pair<string,int>("two", 20),
        pair<string,int>("three", 30),
    };
    map<string, int> object(&pa[0], &pa[3]);
    for (map<string, int>::iterator it = object.begin(); it != object.end(); ++it)
    {
        cout << setw(5) << it->first.c_str() <<
             setw(5) << it->second << '
';
    }
    /*
     * Generated output:
     * one 10
     * three 30
     * two 20
     */
}

12.4.8 The ‘multimap’ container

Like the map, the multimap class implements a (sorted) associative array. Before using a multimap container the header file <map> must be included.

The main difference between the map and the multimap is that the multimap supports multiple values associated with the same key, whereas the map contains single-valued keys. Note that the multimap also accepts multiple identical values associated with identical keys.

The map and the multimap have the same set of constructors and member functions, with the exception of the index operator which is not supported with the multimap. This is understandable: if multiple entries of the same key are allowed, which of the possible values should be returned for object[key]?

Refer to section 12.4.7 for an overview of the multimap member functions. Some member functions, however, deserve additional attention when used in the context of the multimap container. These members are discussed below.

- size_t map::count(key):

  returns the number of entries in the multimap associated with the given key.

- ... erase():

  ...
erases elements from the map:

- `size_t erase(key)` erases all elements having the given `key`. The number of erased elements is returned.
- `void erase(pos)` erases the single element pointed to by `pos`. Other elements possibly having the same keys are not erased.
- `void erase(first, beyond)` erases all elements indicated by the iterator range `[first, beyond)`.

- `pair<multimap::iterator, multimap::iterator> equal_range(key)`:
  returns a pair of iterators, being respectively the return values of `lower_bound` and `upper_bound`, introduced below. The function provides a simple means to determine all elements in the `multimap` that have the same keys. An example illustrating the use of these member functions is given at the end of this section.

- `multimap::iterator find(key)`:
  this member returns an iterator pointing to the first value whose `key` is `key`. If the element isn’t available, `end` is returned. The iterator could be incremented to visit all elements having the same `key` until it is either `end`, or the iterator’s `first` member is not equal to `key` anymore.

- `multimap::iterator insert()`:
  this member function normally succeeds, and so a `multimap::iterator` is returned, instead of a `pair<multimap::iterator, bool>` as returned with the `map` container. The returned iterator points to the newly added element.

Although the functions `lower_bound` and `upper_bound` act identically in the `map` and `multimap` containers, their operation in a `multimap` deserves some additional attention. The next example illustrates `lower_bound`, `upper_bound` and `equal_range` applied to a `multimap`:

```cpp
#include <iostream>
#include <map>
using namespace std;

int main()
{
    pair<string, int> pa[] =
    {
        pair<string, int>("alpha", 1),
        pair<string, int>("bravo", 2),
        pair<string, int>("charley", 3),
        pair<string, int>("bravo", 6),  // unordered 'bravo' values
        pair<string, int>("delta", 5),
        pair<string, int>("bravo", 4),
    };
    multimap<string, int> object(&pa[0], &pa[6]);

typedef multimap<string, int>::iterator msiIterator;

msiIterator it = object.lower_bound("brava");
cout << "Lower bound for 'brava': " <<
```
CHAPTER 12. ABSTRACT CONTAINERS

```cpp
it->first << "", " << it->second << '\n';

it = object.upper_bound("bravu");

cout << "Upper bound for 'bravu': " <<
    it->first << "", " << it->second << '\n';

pair<msiIterator, msiIterator>
    itPair = object.equal_range("bravo");

cout << "Equal range for 'bravo':\n";
for (it = itPair.first; it != itPair.second; ++it)
    cout << it->first << "", " << it->second << '\n';

cout << "Upper bound: " << it->first << "", " << it->second << '\n';

itPair = object.equal_range("brav");

for (it = itPair.first; it != itPair.second; ++it)
    cout << it->first << "", " << it->second << '\n';

cout << "Upper bound: " << it->first << "", " << it->second << '\n';
*/

Generated output:

Lower bound for 'brava': bravo, 2
Upper bound for 'bravu': charley, 3
Equal range for 'bravo':
bravo, 2
bravo, 6
bravo, 4
Upper bound: charley, 3
Equal range for 'brav':
Upper bound: brav, 2
*/

In particular note the following characteristics:

- `lower_bound` and `upper_bound` produce the same result for non-existing keys: they both return the first element having a key that exceeds the provided key.

- Although the keys are ordered in the `multimap`, the values for equal keys are not ordered: they are retrieved in the order in which they were entered.

12.4.9 The ‘set’ container

The `set` class implements a sorted collection of values. Before using `set` containers the `<set>` header file must be included.

A set contains unique values (of a container-acceptable type). Each value is stored only once.

A specific value can be explicitly created: Every `set` defines a `value_type` which may be used to create values that can be stored in the `set`. For example, a value for a `set<string>` can be constructed as follows:
set<string>::value_type setValue("Hello");

The value_type is associated with the set<string>. Anonymous value_type objects are also often used. E.g.,

set<string>::value_type("Hello");

Instead of using the line set<string>::value_type(...) over and over again, a typedef is often used to reduce typing and to improve readability:

typedef set<string>::value_type StringSetValue

Using this typedef, values for the set<string> may be constructed as follows:

StringSetValue("Hello");

Alternatively, values of the set’s type may be used immediately. In that case the value of type Type is implicitly converted to a set<Type>::value_type.

The following constructors, operators, and member functions are available for the set container:

- Constructors:
  - The copy and move constructors are available;
  - A set may be constructed empty:
    ```
    set<int> object;
    ```
  - A set may be initialized using two iterators. For example:
    ```
    int intarr[] = {1, 2, 3, 4, 5};
    set<int> object(&intarr[0], &intarr[5]);
    ```

    Note that all values in the set must be different: it is not possible to store the same value repeatedly when the set is constructed. If the same value occurs repeatedly, only the first instance of the value is entered into the set; the remaining values are silently ignored.

    Like the map, the set receives its own copy of the data it contains.

- The set container only supports the standard set of operators that are available for containers.

- The set class has the following member functions:
  - set::iterator begin():
    ```
    returns an iterator pointing to the first element of the set. If the set is empty end is returned.
    ```
  - void clear():
    ```
    erases all elements from the set.
    ```
  - size_t count(key):
    ```
    returns 1 if the provided key is available in the set, otherwise 0 is returned.
    ```
  - bool empty():
    ```
    returns true if the set contains no elements.
- `set::iterator end()`: returns an iterator pointing beyond the last element of the set.
- `pair<set::iterator, set::iterator> equal_range(key)`: this member returns a pair of iterators, being respectively the return values of the member functions `lower_bound` and `upper_bound`, introduced below.
- `... erase()`: erases a specific element or range of elements from the set:
  * `bool erase(value)` erases the element having the given value from the set. True is returned if the value was removed, false if the set did not contain an element 'value'.
  * `void erase(pos)` erases the element pointed to by the iterator `pos`.
  * `void erase(first, beyond)` erases all elements indicated by the iterator range `[first, beyond)`.
- `set::iterator find(value)`: returns an iterator to the element having the given value. If the element isn't available, `end` is returned.
- `allocator_type get_allocator() const`: returns a copy of the allocator object used by the `set` object.
- `... insert()`: inserts elements into the `set`. If the element already exists, the existing element is left untouched and the element to be inserted is ignored. The return value depends on the version of `insert` that is called:
  * `pair<set::iterator, bool> insert(keyvalue)` inserts a new `set::value_type` into the set. The return value is a `pair<set::iterator, bool>`. If the returned `bool` field is `true`, value was inserted into the set. The value `false` indicates that the value that was specified was already available in the set, and so the provided value was not inserted into the set. In both cases the `set::iterator` field points to the data element in the set having the specified value.
  * `set::iterator insert(pos, keyvalue)`. This way a `set::value_type` may also be inserted into the set. `pos` is ignored, and an iterator to the inserted element is returned.
  * `void insert(first, beyond)` inserts the `set::value_type` elements pointed to by the iterator range `[first, beyond)` into the set.
- `key_compare key_comp()`: returns a copy of the object used by the `set` to compare keys. The type `set<ValueType>::key_compare` is defined by the set container and `key_compare`'s parameters have types `ValueType const &`. The comparison function returns `true` if its first argument should be ordered before its second argument.
- `set::iterator lower_bound(key)`: returns an iterator pointing to the first `keyvalue` element of which the `key` is at least equal to the specified `key`. If no such element exists, the function returns `end`.
- `size_t max_size()`: returns the maximum number of elements this `set` may contain.
- `set::reverse_iterator rbegin()`: returns an iterator pointing to the last element of the set.
12.4. AVAILABLE CONTAINERS

- set::reverse_iterator rend:
  returns an iterator pointing before the first element of the set.

- size_t size():
  returns the number of elements in the set.

- void swap(argument):
  swaps two sets (argument being the second set) that use identical data types.

- set::iterator upper_bound(key):
  returns an iterator pointing to the first keyvalue element having a key exceeding
  the specified key. If no such element exists, the function returns end.

- value_compare value_comp():
  returns a copy of the object used by the set to compare keys. The type
  set< ValueType>::value_compare is defined by the set container and
  value_compare's parameters have types ValueType const &. The compar-
  ison function returns true if its first argument should be ordered before its second
  argument. Its operation is identical to that of a key_compare object, returned by
  key_comp.

12.4.10 The ‘multiset’ container

Like the set, the multiset class implements a sorted collection of values. Before using multiset
containers the header file <set> must be included.

The main difference between the set and the multiset is that the multiset supports multiple
entries of the same value, whereas the set contains unique values.

The set and the multiset have the same set of constructors and member functions. Refer to section
12.4.9 for an overview of the multiset member functions. Some member functions, however, behave
slightly different than their counterparts of the set container. Those members are mentioned here.

- size_t count(value):
  returns the number of entries in the multiset associated with the given value.

- ... erase():
  erases elements from the set:

  - size_t erase(value) erases all elements having the given value. The number of
    erased elements is returned.
  - void erase(pos) erases the element pointed to by the iterator pos. Other elements
    possibly having the same values are not erased.
  - void erase(first, beyond) erases all elements indicated by the iterator range
    [first, beyond).

- pair<multiset::iterator, multiset::iterator> equal_range(value):
  returns a pair of iterators, being respectively the return values of lower_bound and
  upper_bound, introduced below. The function provides a simple means to determine
  all elements in the multiset that have the same values.
• `multiset::iterator find(value)`:
  returns an iterator pointing to the first element having the specified value. If the element isn’t available, `end` is returned. The iterator could be incremented to visit all elements having the given `value` until it is either `end`, or the iterator doesn’t point to ‘`value`’ anymore.

• ... `insert()`:
  this member function normally succeeds and returns a `multiset::iterator` rather than a pair of `multiset::iterator, bool` as returned with the `set` container. The returned iterator points to the newly added element.

Although the functions `lower_bound` and `upper_bound` act identically in the `set` and `multiset` containers, their operation in a `multiset` deserves some additional attention. With a `multiset` container, `lower_bound` and `upper_bound` produce the same result for non-existing keys: they both return the first element having a key exceeding the provided key.

Here is an example showing the use of various member functions of a `multiset`:

```cpp
#include <iostream>
#include <set>

using namespace std;

int main()
{
    string sa[] = {
        "alpha",
        "echo",
        "hotel",
        "mike",
        "romeo"
    };

    multiset<string> object(&sa[0], &sa[5]);

    object.insert("echo");
    object.insert("echo");

    multiset<string>::iterator it = object.find("echo");

    for (; it != object.end(); ++it)
    {
        cout << *it << " ";
        cout << '\n';
    }

    cout << "Multiset::equal_range(\"{e}\")\n";
    pair <
        multiset<string>::iterator,
        multiset<string>::iterator
```
12.4. AVAILABLE CONTAINERS

```cpp
> itpair = object.equal_range("ech");

if (itpair.first != object.end())
    cout << "lower_bound() points at " << *itpair.first << '\n';
for (; itpair.first != itpair.second; ++itpair.first)
    cout << *itpair.first << " ";
cout << '\n' << object.count("ech") << " occurrences of 'ech' " << '\n';

cout << "Multiset::equal_range(+latexcommand("{e})cho")\n";
    itpair = object.equal_range("echo");
for (; itpair.first != itpair.second; ++itpair.first)
    cout << *itpair.first << " ";
    cout << '\n' << object.count("echo") << " occurrences of 'echo' " << '\n';
    cout << "Multiset::equal_range(+latexcommand("{e})choo")\n";
        itpair = object.equal_range("echoo");
    for (; itpair.first != itpair.second; ++itpair.first)
        cout << *itpair.first << " ";
    cout << '\n' << object.count("echoo") << " occurrences of 'echoo' " << '\n';
```

Generated output:

```
echo echo echo hotel mike romeo
Multiset::equal_range("ech")
lower_bound() points at echo

0 occurrences of 'ech'
Multiset::equal_range("echo")
echo echo echo
3 occurrences of 'echo'
Multiset::equal_range("echoo")

0 occurrences of 'echoo'

*/
```

12.4.11 The ‘stack’ container

The `stack` class implements a stack data structure. Before using `stack` containers the header file `<stack>` must be included.

A stack is also called a first in, last out (FILO or LIFO) data structure as the first item to enter the stack is the last item to leave. A stack is an extremely useful data structure in situations where data must temporarily remain available. For example, programs maintain a stack to store local
variables of functions: the lifetime of these variables is determined by the time these functions are active, contrary to global (or static local) variables, which live for as long as the program itself lives. Another example is found in calculators using the Reverse Polish Notation (RPN), in which the operands of operators are kept in a stack, whereas operators pop their operands off the stack and push the results of their work back onto the stack.

As an example of the use of a stack, consider figure 12.5, in which the contents of the stack is shown while the expression \((3 + 4) \times 2\) is evaluated. In the RPN this expression becomes \(3 \ 4 + 2 \times\), and figure 12.5 shows the stack contents after each token (i.e., the operands and the operators) is read from the input. Notice that each operand is indeed pushed on the stack, while each operator changes the contents of the stack. The expression is evaluated in five steps. The caret between the tokens in the expressions shown on the first line of figure 12.5 shows what token has just been read. The next line shows the actual stack-contents, and the final line shows the steps for referential purposes. Note that at step 2, two numbers have been pushed on the stack. The first number (3) is now at the bottom of the stack. Next, in step 3, the + operator is read. The operator pops two operands (so that the stack is empty at that moment), calculates their sum, and pushes the resulting value (7) on the stack. Then, in step 4, the number 2 is read, which is dutifully pushed on the stack again. Finally, in step 5 the final operator \(\times\) is read, which pops the values 2 and 7 from the stack, computes their product, and pushes the result back on the stack. This result (14) could then be popped to be displayed on some medium.

From figure 12.5 we see that a stack has one location (the top) where items can be pushed onto and popped off the stack. This top element is the stack’s only immediately visible element. It may be accessed and modified directly.

Bearing this model of the stack in mind, let’s see what we formally can do with the stack container. For the stack, the following constructors, operators, and member functions are available:

- **Constructors:**
  - The copy and move constructors are available;
  - A stack may be constructed empty:
    ```
    stack<string> object;
    ```

- Only the basic set of container operators are supported by the stack

- The following member functions are available for stacks:
12.4. AVAILABLE CONTAINERS

- bool empty():
  this member returns true if the stack contains no elements.
- void pop():
  removes the element at the top of the stack. Note that the popped element is not returned by this member, and refer to section 12.4.4 for a discussion about the reason why pop has return type void.

Furthermore, it is the responsibility of the stack's user to assure that pop is not called when the stack is empty. If pop is called for an empty stack, its internal administration breaks, resulting, e.g., in a negative size (showing itself as a very large stacksize due to its size member returning a size_t, and other operations (like push) fail and may crash your program. Of course, with a well designed algorithm requests to pop from empty stacks do not occur (which is probably why this implementation was used for the stack container).

- void push(value):
  places value at the top of the stack, hiding the other elements from view.
- size_t size():
  this member returns the number of elements in the stack.
- Type &top():
  this member returns a reference to the stack's top (and only visible) element. It is the responsibility of the programmer to use this member only if the stack is not empty.

The stack does not support iterators or a subscript operator. The only elements that can be accessed is its top element. To empty a stack:

- repeatedly remove its front element;
- assign an empty stack to it;
- have its destructor called (e.g., by ending its lifetime).

12.4.12 The ‘unordered_map’ container (‘hash table’)

The C++11 standard officially adds hash tables (the unordered_map) to the language.

Before using unordered_map or unordered_multimap containers the header file <unordered_map> must be included.

The unordered_map class implements an associative array in which the elements are stored according to some hashing scheme. As discussed, the map is a sorted data structure. The keys in maps are sorted using the operator< of the key's data type. Generally, this is not the fastest way to either store or retrieve data. The main benefit of sorting is that a listing of sorted keys appeals more to humans than an unsorted list. However, a by far faster way to store and retrieve data is to use hashing.

Hashing uses a function (called the hash function) to compute an (unsigned) number from the key, which number is thereupon used as an index in the table storing the keys and their values. This number is called the bucket number. Retrieval of a key is as simple as computing the hash value of the provided key, and looking in the table at the computed index location: if the key is present, it is stored in the table, at the computed bucket location and its value can be returned. If it's not present, the key is not currently stored in the container.
Collisions occur when a computed index position is already occupied by another element. For these situations the abstract containers have solutions available. A simple solution, adopted by the C++11 standard is to use linear chaining which uses linked list to store colliding table elements in.

In the C++11 standard the term unordered_map is used rather than hash to avoid name collisions with hash tables developed before the C++11 standard was established.

Because of the hashing method, the efficiency of a unordered_map in terms of speed should greatly exceed the efficiency of the map. Comparable conclusions may be drawn for the unordered_set, the unordered_multimap and the unordered_multiset.

### 12.4.12.1 The ‘unordered_map’ constructors

When defining an unordered_map type five template arguments must be specified:

- a KeyType (becoming unordered_map::key_type),
- a ValueType (becoming unordered_map::mapped_type),
- the type of an object computing a hash value from a key value (becoming unordered_map::hasher), and
- the type of an object that can compare two keys for equality (becoming unordered_map::key_equal).
- the type of its allocator. This is usually left unspecified, using the allocator provided by default by the implementor.

The generic definition of an unordered_map container looks like this:

```cpp
std::unordered_map<KeyType, ValueType, hash_type, predicate_type, allocator_type>
```

When KeyType is std::string or a built-in type then default types are available for the hash type and the predicate type. In practice the allocator type is not specified, as the default allocator suffices. In these cases an unordered_map object can be defined by merely specifying the key- and value types, like this:

```cpp
std::unordered_map<std::string, ValueType> hash(size_t size = implSize);
```

Here, implSize is the container's default initial size, which is specified by the implementor. The map's size is automatically enlarged by the unordered_map when necessary, in which case the container rehashes all its elements. In practice the default size argument provided by the implementor is completely satisfactory.

The KeyType frequently consists of text. So, a unordered_map using a std::string KeyType is frequently used. Be careful not to use a plain char const * key_type as two char const * values pointing to equal C-strings stored at different locations are considered to be different keys, as their pointer values rather than their textual contents are compared. Here is an example showing how a char const * KeyType can be used. Note that in the example no arguments are specified when constructing months, since default values and constructors are available:

```cpp
#include <unordered_map>
```
12.4. AVAILABLE CONTAINERS

```cpp
#include <iostream>
#include <string>
#include <cstring>

using namespace std;

struct EqualCp
{
    bool operator()(char const *l, char const *r) const
    {
        return strcmp(l, r) == 0;
    }
};
struct HashCp
{
    size_t operator()(char const *str) const
    {
        return std::hash<std::string>()(str);
    }
};

int main()
{
    unordered_map<char const *, int, HashCp, EqualCp> months;
    // or explicitly:
    unordered_map<char const *, int, HashCp, EqualCp> monthsTwo(61, HashCp(), EqualCp());

    months["april"] = 30;
    months["november"] = 31;

    string apr("april"); // different pointers, same string
    cout << "april -> " << months["april"] << \n' <<
    "april -> " << months[apr.c_str()] << \n';
}
```

If other KeyTypes must be used, then the unordered_map's constructor requires (constant references to) a hash function object, computing a hash value from a key value, and a predicate function object, returning `true` if two unordered_map::key_type objects are identical. A generic algorithm (see chapter 19) exists performing tests of equality (i.e., `equal_to`). These tests can be used if the key's data type supports the equality operator. Alternatively, an overloaded operator== or specialized function object could be constructed returning `true` if two keys are equal and `false` otherwise.

Constructors

The unordered_map supports the following constructors:

- The copy and move constructors are available;
- `explicit unordered_map(size_type n = implSize, hasher const &hf = hasher(),
  key_equal const &eql = key_equal(),
  allocator_type const &alloc = allocator_type()):` this constructor can also be used as default constructor;
• `unordered_map(const_iterator begin, const_iterator end, size_type n = implSize, hasher const &hf = hasher(), key_equal const &eql = key_equal(), allocator_type const &alloc = allocator_type());` this constructor expects two iterators specifying a range of unordered_map::value_type const objects, and

• `unordered_map(initializer_list<value_type> initList, size_type n = implSize, hasher const &hf = hasher(), key_equal const &eql = key_equal(), allocator_type const &alloc = allocator_type());` a constructor expecting an initializer_list of unordered_map::value_type values.

The following example shows a program using an unordered_map containing the names of the months of the year and the number of days these months (usually) have. Then, using the subscript operator the days in several months are displayed (the predicate used here is the generic algorithm `equal_to<string>`, which is provided by the compiler as the default fourth argument of the unordered_map constructor):

```cpp
#include <unordered_map>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    unordered_map<string, int> months;
    months["january"] = 31;
    months["february"] = 28;
    months["march"] = 31;
    months["april"] = 30;
    months["may"] = 31;
    months["june"] = 30;
    months["july"] = 31;
    months["august"] = 31;
    months["september"] = 30;
    months["october"] = 31;
    months["november"] = 30;
    months["december"] = 31;

    cout << "september -> " << months["september"] << 'n' <<
         "april -> " << months["april"] << 'n' <<
         "june -> " << months["june"] << 'n' <<
         "november -> " << months["november"] << 'n';
}
/*
   Generated output:
   september -> 30
   april -> 30
   june -> 30
   november -> 30
*/
```
12.4. AVAILABLE CONTAINERS

12.4.12.2 The ‘unordered_map’ public members

The unordered_map supports the index operator operating identically to the map’s index operator: a (const) reference to the ValueType associated with the providedKeyType’s value is returned. If not yet available, the key is added to the unordered_map, and a default ValueType value is returned. In addition, it supports operator==.

The unordered_map provides the following member functions (key_type, value_type etc. refer to the types defined by the unordered_map):

- mapped_type &at(key_type const &key):
  returns a reference to the unordered_map’s mapped_type associated with key. If the key is not stored in the unordered_map an std::out_of_range exception is thrown.
- unordered_map::iterator begin():
  returns an iterator pointing to the first element in the unordered_map, returning end if the unordered_map is empty.
- size_t bucket(key_type const &key):
  returns the index location where key is stored. If key wasn’t stored yet bucket adds value_type(key, Value()) before returning its index position.
- size_t bucket_count():
  returns the number of slots used by the containers. Each slot may contain one (or more, in case of collisions) value_type objects.
- size_t bucket_size(size_t index):
  returns the number of value_type objects stored at bucket position index.
- unordered_map::const_iterator cbegin():
  returns a const_iterator pointing to the first element in the unordered_map, returning cend if the unordered_map is empty.
- unordered_map::const_iterator cend():
  returns a const_iterator pointing just beyond the unordered_map’s last element.
- void clear():
  erases all the unordered_map’s elements.
- size_t count(key_type const &key):
  returns the number of times a value_type object using key_type key is stored in the unordered_map (which is either one or zero).
- pair<iterator, bool> emplace(Args &&...args):
  a value_type object is constructed from emplace’s arguments. If the unordered_map already contained an object using the same key_type value, then a std::pair is returned containing an iterator pointing to the object using the same key_type value and the value false. If no such key_type value was found, the newly constructed object is inserted into the unordered_map, and the returned std::pair contains an iterator pointing to the newly inserted value_type as well as the value true.
• iterator emplace_hint(const_iterator position, Args &&...args):
  a value_type object is constructed from the member's arguments, and the newly
  created element is inserted into the unordered_map, unless the (at args) pro-
  vided key already exists. The implementation may or may not use position
  as a hint to start looking for an insertion point. The returned iterator points
  to the value_type using the provided key. It may refer to an already existing
  value_type or to a newly added value_type; an existing value_type is not re-
  placed. If a new value was added, then the container's size has been incremented
  when emplace_hint returns.

• bool empty():
  returns true if the unordered_map contains no elements.

• unordered_map::iterator end():
  returns an iterator pointing beyond the last element in the unordered_map.

• pair<iterator, iterator> equal_range(key):
  this member returns a pair of iterators defining the range of elements having a key
  that is equal to key. With the unordered_map this range includes at most one
  element.

• unordered_map::iterator erase():
  erases a specific range of elements in the unordered_map:
  - erase(pos) erases the element pointed to by the iterator pos. The iterator ++pos is
    returned.
  - erase(first, beyond) erases elements indicated by the iterator range [first,
    beyond), returning beyond.

• iterator find(key):
  returns an iterator to the element having the given key. If the element isn’t available,
  end is returned.

• allocator_type get_allocator() const:
  returns a copy of the allocator object used by the unordered_map object.

• hasher hash_function() const:
  returns a copy of the hash function object used by the unordered_map object.

• ... insert():
  elements may be inserted starting at a certain position. No insertion is performed
  if the provided key is already in use. The return value depends on the version of
  insert() that is called. When a pair<iterator, bool> is returned, then the
  pair’s first member is an iterator pointing to the element having a key that is
  equal to the key of the provided value_type, the pair’s second member is true
  if value was actually inserted into the container, and false if not.
  - pair<iterator, bool> insert(value_type const &value) attempts to insert
    value.
  - pair<iterator, bool> insert(value_type &&tmp) attempts to insert value us-
    ing value_type's move constructor.
12.4. AVAILABLE CONTAINERS

- `pair<iterator, bool> insert(const_iterator hint, value_type const &value)` attempts to insert `value`, possibly using `hint` as a starting point when trying to insert `value`.

- `pair<iterator, bool> insert(const_iterator hint, value_type &&tmp)` attempts to insert a `value` using `value_type`'s move constructor, and possibly using `hint` as a starting point when trying to insert `value`.

- `void insert(first, beyond)` tries to insert the elements in the iterator range `[first, beyond)`.

- `void insert(initializer_list <value_type> iniList)` attempts to insert the elements in `iniList` into the container.

- `hasher key_eq() const`:
  returns a copy of the `key_equal` function object used by the `unordered_map` object.

- `float load_factor() const`:
  returns the container's current load factor, i.e. `size / bucket_count`.

- `size_t max_bucket_count()`:
  returns the maximum number of buckets this `unordered_map` may contain.

- `float max_load_factor() const`:
  identical to `load_factor`.

- `void max_load_factor(float max)`:
  changes the current maximum load factor to `max`. When a load factor of `max` is reached, the container will enlarge its `bucket_count`, followed by a rehash of its elements. Note that the container's default maximum load factor equals 1.0

- `size_t max_size()`:
  returns the maximum number of elements this `unordered_map` may contain.

- `void rehash(size_t size)`:
  if `size` exceeds the current bucket count, then the bucket count is increased to `size`, followed by a rehash of its elements.

- `void reserve(size_t request)`:
  if `request` is less than or equal to the current bucket count, this call has no effect. Otherwise, the bucket count is increased to a value of at least `request`, followed by a rehash of the container's elements.

- `size_t size()`:
  returns the number of elements in the `unordered_map`.

- `void swap(unordered_map &other)`:
  swaps the contents of the current and the other `unordered_map`. 
12.4.12.3 The ‘unordered_multimap’ container

The `unordered_multimap` allows multiple objects using the same keys to be stored in an unordered map. The `unordered_multimap` container offers the same set of members and constructors as the `unordered_map`, but without the unique-key restriction imposed upon the `unordered_map`.

The `unordered_multimap` does not offer `operator[]` and does not offer the `at` members.

Below all members are described whose behavior differs from the behavior of the corresponding `unordered_map` members:

- **at**
  
  not supported by the `unordered_multimap` container

- **size_t count(key_type const &key):**
  
  returns the number of times a `value_type` object using `key_type key` is stored in the `unordered_map`. This member is commonly used to verify whether `key` is available in the `unordered_multimap`.

- **iterator emplace(Args &&...args):**
  
  a `value_type` object is constructed from `emplace`'s arguments. The returned iterator points to the newly inserted `value_type`.

- **iterator emplace_hint(const_iterator position, Args &&...args):**
  
  a `value_type` object is constructed from the member's arguments, and the newly created element is inserted into the `unordered_multimap`. The implementation may or may not use position as a hint to start looking for an insertion point. The returned iterator points to the `value_type` using the provided key.

- **pair<iterator, iterator> equal_range(key):**
  
  this member returns a pair of iterators defining the range of elements having a key that is equal to `key`.

- **iterator find(key):**
  
  returns an iterator to an element having the given key. If no such element is available, `end` is returned.

- **... insert():**
  
  elements may be inserted starting at a certain position. The return value depends on the version of `insert()` that is called. When an iterator is returned, then it points to the element that was inserted.

  - `iterator insert(value_type const &value)` inserts value.
  
  - `iterator insert(value_type &&tmp)` inserts value using `value_type`'s move constructor.
  
  - `iterator insert(const_iterator hint, value_type const &value)` inserts value, possibly using hint as a starting point when trying to insert value.
  
  - `iterator insert(const_iterator hint, value_type &&tmp)` inserts value using `value_type`'s move constructor, and possibly using hint as a starting point when trying to insert value.
12.4. AVAILABLE CONTAINERS

- void insert(first, beyond) inserts the elements in the iterator range [first, beyond).
- void insert(initializer_list <value_type> iniList) inserts the elements in iniList into the container.

12.4.13 The ‘unordered_set’ container

The set container, like the map container, orders its elements. If ordering is not an issue, but fast lookups are, then a hash-based set and/or multi-set may be preferred. The C++ standard has added such hash-based sets and multi-sets to the language: the unordered_set and unordered_multiset.

Before using these hash-based set containers the header file <unordered_set> must be included.

Elements stored in the unordered_set are immutable, but they can be inserted and removed from the container. Different from the unordered_map, the unordered_set does not use a ValueType. The set merely stores elements, and the stored element itself is its own key.

The unordered_set has the same constructors as the unordered_map, but the set’s value_type is equal to its key_type.

When defining an unordered_set type four template arguments must be specified:

- a KeyType (becoming unordered_set::key_type),
- the type of an object computing a hash value from a key value (becoming unordered_set::hasher), and
- the type of an object that can compare two keys for equality (becoming unordered_set::key_equal).
- the type of its allocator. This is usually left unspecified, using the allocator provided by default by the implementor.

The generic definition of an unordered_set container looks like this:

```
std::unordered_set <KeyType, hash type, predicate type, allocator type>
```

When KeyType is std::string or a built-in type then default types are available for the hash type and the predicate type. In practice the allocator type is not specified, as the default allocator suffices. In these cases an unordered_set object can be defined by merely specifying the key- and value types, like this:

```
std::unordered_set<std::string> rawSet(size_t size = implSize);
```

Here, implSize is the container’s default initial size, which is specified by the implementor. The set’s size is automatically enlarged when necessary, in which case the container rehashes all its elements. In practice the default size argument provided by the implementor is completely satisfactory.

The unordered_set supports the following constructors:

- The copy and move constructors are available;
• explicit unordered_set(size_type n = implSize, hasher const &hf = 
  hasher(), 
  key_equal const &eql = key_equal(), 
  allocator_type const &alloc = allocator_type()): this constructor can also be 
  used as default constructor;

• unordered_set(const_iterator begin, const_iterator end, size_type 
  n = implSize, hasher const &hf = hasher(), key_equal const &eql = 
  key_equal(), allocator_type const &alloc = allocator_type()): this con-
  structor expects two iterators specifying a range of unordered_set::value_type 
  const objects, and

• unordered_set(initializer_list<value_type> initList, size_type n 
  = implSize, hasher const &hf = hasher(), key_equal const &eql = 
  key_equal(), allocator_type const &alloc = allocator_type()): a constructor 
  expecting an initializer_list of unordered_set::value_type values.

The unordered_set does not offer an index operator, and it does not offer an at member. Other 
than those, it offers the same members as the unordered_map. Below the members whose behavior 
differs from the behavior of the unordered_map are discussed. For a description of the remaining 
members, please refer to section 12.4.12.2.

• iterator emplace(Args &&...args):
  a value_type object is constructed from emplace’s arguments. It is added to the 
  set if it is unique, and an iterator to the value_type is returned.

• iterator emplace_hint(const_iterator position, Args &&...args):
  a value_type object is constructed from the member’s arguments, and if the newly 
  created element is unique it is inserted into the unordered_set. The implementa-
  tion may or may not use position as a hint to start looking for an insertion point. 
  The returned iterator points to the value_type.

• unordered_set::iterator erase():
  erases a specific range of elements in the unordered_set:

  - erase(key_type const &key) erases key from the set. An iterator pointing to the 
    next element is returned.
  - erase(pos) erases the element pointed to by the iterator pos. The iterator ++pos is 
    returned.
  - erase(first, beyond) erases elements indicated by the iterator range [first, 
    beyond), returning beyond.

12.4.13.1 The ‘unordered_multiset’ container

The unordered_multiset allows multiple objects using the same keys to be stored in an unordered 
set. The unordered_multiset container offers the same set of members and constructors as the 
unordered_set, but without the unique-key restriction imposed upon the unordered_set.

Below all members are described whose behavior differs from the behavior of the corresponding 
unordered_set members:
• `size_t count(key_type const &key)`:
  returns the number of times a `value_type` object using `key_type` key is stored in the `unordered_set`. This member is commonly used to verify whether key is available in the `unordered_multiset`.

• `iterator emplace(Args &&...args)`:
  a `value_type` object is constructed from `emplace`'s arguments. The returned iterator points to the newly inserted `value_type`.

• `iterator emplace_hint(const_iterator position, Args &&...args)`:
  a `value_type` object is constructed from the member's arguments, and the newly created element is inserted into the `unordered_multiset`. The implementation may or may not use `position` as a `hint` to start looking for an insertion point. The returned iterator points to the `value_type` using the provided key.

• `pair<iterator, iterator> equal_range(key)`:
  this member returns a pair of iterators defining the range of elements having a key that is equal to `key`.

• `iterator find(key)`:
  returns an iterator to an element having the given key. If no such element is available, `end` is returned.

• ... insert():
  elements may be inserted starting at a certain position. The return value depends on the version of `insert()` that is called. When an `iterator` is returned, then it points to the element that was inserted.

  - `iterator insert(value_type const &value)` inserts `value`.
  - `iterator insert(value_type &&tmp)` inserts `value` using `value_type`'s move constructor.
  - `iterator insert(const_iterator hint, value_type const &value)` inserts `value`, possibly using `hint` as a starting point when trying to insert `value`.
  - `iterator insert(const_iterator hint, value_type &&tmp)` inserts `value` using `value_type`'s move constructor, and possibly using `hint` as a starting point when trying to insert `value`.
  - `void insert(first, beyond)` inserts the elements in the iterator range `[first, beyond)`.
  - `void insert(initializer_list <value_type> iniList)` inserts the elements in `iniList` into the container.

12.5 The ‘complex’ container

The `complex` container defines the standard operations that can be performed on complex numbers. Before using `complex` containers the header file `<complex>` must be included.

The complex number's real and imaginary types are specified as the container’s data type. Examples:

```
complex<double>
complex<int>
complex<float>
```
Note that the real and imaginary parts of complex numbers have the same datatypes.

When initializing (or assigning) a complex object, the imaginary part may be omitted from the initialization or assignment resulting in its value being 0 (zero). By default, both parts are zero.

Below it is silently assumed that the used complex type is `complex<double>`. Given this assumption, complex numbers may be initialized as follows:

- **target**: A default initialization: real and imaginary parts are 0.
- **target(1)**: The real part is 1, imaginary part is 0
- **target(0, 3.5)**: The real part is 0, imaginary part is 3.5
- **target(source)**: `target` is initialized with the values of `source`.

Anonymous complex values may also be used. In the next example two anonymous complex values are pushed on a stack of complex numbers, to be popped again thereafter:

```cpp
#include <iostream>
#include <complex>
#include <stack>

using namespace std;

int main()
{
    stack<complex<double>> cstack;

    cstack.push(complex<double>(3.14, 2.71));
    cstack.push(complex<double>(-3.14, -2.71));

    while (cstack.size())
    {
        cout << cstack.top().real() <<", " << 
            cstack.top().imag() << "i" << 'n';
        cstack.pop();
    }
    /*
     * Generated output:
     * -3.14, -2.71i
     * 3.14, 2.71i
     */
}
```

The following member functions and operators are defined for complex numbers (below, `value` may be either a primitive scalar type or a complex object):

- Apart from the standard container operators, the following operators are supported from the complex container.
  - `complex operator+(value)`:
    this member returns the sum of the current complex container and `value`. 
12.6. UNRESTRICTED UNIONS

- complex operator-(value):
  this member returns the difference between the current complex container and value.
- complex operator*(value):
  this member returns the product of the current complex container and value.
- complex operator/(value):
  this member returns the quotient of the current complex container and value.
- complex operator+=(value):
  this member adds value to the current complex container, returning the new value.
- complex operator-=(value):
  this member subtracts value from the current complex container, returning the new value.
- complex operator*=(value):
  this member multiplies the current complex container by value, returning the new value.
- complex operator/=(value):
  this member divides the current complex container by value, returning the new value.

- Type real():
  returns the real part of a complex number.
- Type imag():
  returns the imaginary part of a complex number.

- Several mathematical functions are available for the complex container, such as abs, arg, conj, cos, cosh, exp, log, norm, polar, pow, sin, sinh and sqrt. All these functions are free functions, not member functions, accepting complex numbers as their arguments. For example,

  abs(complex<double>(3, -5));
  pow(target, complex<int>(2, 3));

- Complex numbers may be extracted from istream objects and inserted into ostream objects. The insertion results in an ordered pair \((x, y)\), in which \(x\) represents the real part and \(y\) the imaginary part of the complex number. The same form may also be used when extracting a complex number from an istream object. However, simpler forms are also allowed. E.g., when extracting 1.2345 the imaginary part is set to 0.

12.6 Unrestricted Unions

We end this chapter about abstract containers with a small detour, introducing additions to the union concept, made available by the C++11 standard. Although unions themselves aren’t ‘abstract containers’, having covered containers has put us in a good position to introduce and illustrate unrestricted unions.

The C++11 standard adds unrestricted unions to C++’s data structuring capabilities. Whereas the traditional union can only contain primitive data, unrestricted unions allow data fields of types for
which non-trivial constructors have been defined. Such data fields commonly are of class-types. Here is an example of such an unrestricted union:

```cpp
union Union
{
    int u_int;
    std::string u_string;
};
```

One of its fields defines a constructor, turning this union into an unrestricted union. As an unrestricted union defines at least one field of a type having a constructor the question becomes how these unions can be constructed and destroyed.

The destructor of a union consisting of, e.g. a `std::string` and an `int` should of course not call the string's destructor if the union's last (or only) use referred to its `int` field. Likewise, when the `std::string` field is used, and a switch is made next from the `std::string` to the `int` field, `std::string`'s destructor should be called before any assignment to the `double` field takes place.

The compiler does not solve the issue for us, and in fact does not implement default constructors or destructors for unrestricted unions at all. If we try to define an unrestricted union like the one shown above, an error message like the following is issued:

```
error: use of deleted function 'Union::Union()'
error: 'Union::Union()' is implicitly deleted because the default definition would be ill-formed:
error: union member 'Union::u_string' with non-trivial '
  std::basic_string<...>::basic_string()' ...
```

### 12.6.1 Implementing the destructor

Although the compiler won't provide (default) implementations for constructors and destructors of unrestricted unions, we can. The task isn't difficult, but there are some caveats.

Consider our unrestricted union's destructor. It clearly should destroy `u_string`'s data if that is its currently active field; but it should do nothing if `u_int` is its currently active field. But how does the destructor know what field to destroy? It doesn't as the unrestricted union holds no information about what field is currently active.

Here is one way to tackle this problem:

If we embed the unrestricted union in a larger aggregate, like a class or a struct, then the class or struct can be provided with a `tag` data member storing the currently active union-field. The tag can be an enumeration type, defined by the aggregate. The unrestricted union may then be controlled by the aggregate. Under this approach we start out with an explicit empty implementations of the destructor, as there's no way to tell the destructor itself what field to destroy:

```cpp
Data::Union::~Union()
{};
```

### 12.6.2 Embedding an unrestricted union in a surrounding class

Next, we embed the unrestricted union in a surrounding aggregate: class `Data`. The aggregate is provided with an enum `Tag`, declared in its public section, so `Data`'s users may request tags. `Union`
12.6. UNRESTRICTED UNIONS

itself is for Data’s internal use only, so Union is declared in Data’s private section. Using a struct Data rather than class Data we start out in a public section, saving us from having to specify the initial public: section for enum Tag:

```cpp
struct Data {
    enum Tag {
        INT,
        STRING
    };

    private:
    union Union {
        int u_int;
        std::string u_string;
        ~Union(); // no actions
        // ... to do: declarations of members
    };

    Tag d_tag;
    Union d_union;
};
```

Data’s constructors receive int or string values. To pass these values on to d_union, we need Union constructors for the various union fields; matching Data constructors also initialize d_tag to proper values:

```cpp
Data::Union::Union(int value) :
    u_int(value)
{}
Data::Union::Union(std::string const &str) :
    u_string(str)
{}
Data::Data(std::string const &str) :
    d_tag(STRING),
    d_union(str)
{}
Data::Data(int value) :
    d_tag(INT),
    d_union(value)
{}
12.6.3 Destroying an embedded unrestricted union

Data's destructor has a data member which is an unrestricted union. As the union's destructor can't perform any actions, the union's proper destruction is delegated to a member, Union::destroy destroying the fields for which destructors are defined. As d_tag stores the currently used Union field, Data's destructor can pass d_tag to Union::destroy to inform it about which field should be destroyed.

Union::destroy does not need to perform any action for INT tags, but for STRING tags the memory allocated by u_string must be returned. For this an explicit destructor call is used. Union::destroy and Data's destructor are therefore implemented like this:

```cpp
void Data::Union::destroy(Tag myTag)
{
    if (myTag == Tag::STRING)
        u_string.~string();
}

Data::~Data()
{
    d_union.destroy(d_tag);
}
```

12.6.4 Copy and move constructors

Union's copy and move constructors suffer from the same problem as Union's destructor does: the union does not know which is its currently active field. But again: Data does, and by defining 'extended' copy and move constructors, also receiving a Tag argument, these extended constructors can perform their proper initializations. The Union's copy- and move-constructors are deleted, and extended copy- and move constructors are declared:

```cpp
Union(Union const &other) = delete;
Union &operator=(Union const &other) = delete;

Union(Union const &other, Tag tag);
Union(Union &&tmp, Tag tag);
```

Shortly we'll encounter a situation where we must be able to initialize a block of memory using an existing Union object. This task can be performed by copy members, whose implementations are trivial, and which may be used by the above constructors. They can be declared in Union's private section, and have identical parameter lists as the above constructors:

```cpp
void copy(Union const &other, Tag tag);
void copy(Union &&other, Tag tag);
```

The constructors merely have to call these copy members:

```cpp
inline Data::Union::Union(Union const &other, Tag tag)
{
    copy(other, tag);
}
```
12.6. UNRESTRICTED UNIONS

```cpp
inline Data::Union::Union(Union &&tmp, Tag tag)
{
    copy(std::move(tmp), tag);
}
```

Interestingly, no ‘initialization followed by assignment’ happens here: `d_union` has not been initialized in any way by the time we reach the statement blocks of the above constructors. But upon reaching the statement blocks, `d_union` memory is merely raw memory. This is no problem, as the `copy` members use placement new to initialize the `Union`'s memory:

```cpp
void Data::Union::copy(Union const &other, Tag otag)
{
    if (tag == INT)
        u_int = other.u_int;
    else
        new (this) string(other.u_string);
}
```

```cpp
void Data::Union::copy(Union &&tmp, Tag tag)
{
    if (tag == INT)
        u_int = tmp.u_int;
    else
        new (this) string(std::move(tmp.u_string));
}
```

12.6.5 Assignment

To assign a `Data` object to another data object, we need an assignment operator. The standard mold for the assignment operator looks like this:

```cpp
Class &Class::operator=(Class const &other)
{
    Class tmp(other);
    swap(*this, tmp);
    return *this;
}
```

This implementation is exception safe: it offers the ‘commit or roll-back’ guarantee (cf. section 9.6). But can it be applied to `Data`?

It depends. It depends on whether `Data` objects can be fast swapped (cf. section 9.6.1.1) or not. If `Union`'s fields can be fast swapped then we can simply swap bytes and we’re done. In that case `Union` does not require any additional members (to be specific: it won’t need an assignment operator).

But now assume that `Union`'s fields cannot be fast swapped. How to implement an exception-safe assignment (i.e., an assignment offering the ‘commit or roll-back’ guarantee) in that case? The `d_tag` field clearly isn’t a problem, so we delegate the responsibility for proper assignment to `Union`, implementing `Data`'s assignment operators as follows:

```cpp
Data &Data::operator=(Data const &rhs)
```
{  
  if (d_union.assign(d_tag, rhs.d_union, rhs.d_tag))  
    d_tag = rhs.d_tag;  
  return *this;  
}

Data &Data::operator=(Data &&tmp)  
{  
  if (d_union.assign(d_tag, std::move(tmp.d_union), tmp.d_tag))  
    d_tag = tmp.d_tag;  
  return *this;  
}

But now for Union::assign. Assuming that both Unions use different fields, but swapping objects of the separate types is allowed. Now things may go wrong. Assume the left-side union uses type X, the right-side union uses type Y and both types use allocation. First, briefly look at standard swapping. It involves three steps:

- tmp(lhs): initialize a temporary object;
- lhs = rhs: assign the rhs object to the lhs object;
- rhs = tmp: assign the tmp object to the rhs

Usually we assume that these steps do not throw exceptions, as swap itself shouldn’t throw exceptions. How could we implement swapping for our union? Assume the fields are known (easily done by passing Tag values to Union::swap):

- X tmp(lhs.x): initialize a temporary X;
- in-place destroy lhs.x; placement new initialize lhs.y from rhs.y (alternatively: placement new default initialize lhs.y, then do the standard lhs.y = rhs.y)
- in-place destroy rhs.y; placement new initialize rhs.x from tmp (alternatively: placement new default initialize rhs.x, then do the standard rhs.x = tmp)

By C++-standard requirement, the in-place destruction won’t throw. Since the standard swap also performs an assignment that part should work fine as well. And since the standard swap also does a copy construction the placement new operations should perform fine as well, and if so, Union may be provided with the following swap member:

```cpp
void Data::Union::swap(Tag myTag, Union &other, Tag oTag)  
{  
  Union tmp(*this, myTag); // save lhs  
  destroy(myTag); // destroy lhs  
  copy(other, oTag); // assign rhs  
  other.destroy(oTag); // destroy rhs  
  other.copy(tmp, myTag); // save lhs via tmp  
}
```

Now that swap is available Data’s assignment operators are easily realized:

```cpp
Data &Data::operator=(Data const &rhs)
```
12.6. UNRESTRICTED UNIONS

```cpp
Data tmp(rhs); // tmp(std::move(rhs)) for the move assignment
d_union.swap(d_tag, tmp.d_union, tmp.d_tag);
swap(d_tag, tmp.d_tag);
return *this;
}
```

What if the `Union` constructors could throw? In that case we can provide `Data` with an 'commit or roll-back' assignment operator like this:

```cpp
Data &Data::operator=(Data const &rhs)
{
    Data tmp(rhs);
    // rolls back before throwing an exception
d_union.assign(d_tag, rhs.d_union, rhs.d_tag);
d_tag = rhs.d_tag;
    return *this;
}
```

How to implement `Union::assign`? Here are the steps `assign` must take:

- First save the current union in a block of memory. This merely involves a non-throwing `memcpy` operation;
- Then use placement new to copy the other object's union field into the current object. If this throws:
  - catch the exception, restore the original `Union` from the saved block and rethrow the exception: we have rolled-back to our previous (valid) state.
- We still have to delete the original field's allocated data. To do so, we perform the following steps:
  - (Fast) swap the current union's new contents with the contents in the previously saved block;
  - Call `destoy` for the now restored original union;
  - Re-install the new union from the memory block.

As none of the above steps will throw, we have committed the new situation.

Here is the implementation of the 'commit or roll-back' `Union::assign`:

```cpp
void Data::Union::assign(Tag myTag, Union const &other, Tag otag)
{
    char saved[sizeof(Union)];
    memcpy(saved, this, sizeof(Union)); // raw copy: saved <- *this
    try
    {
        copy(other, otag); // *this = other: may throw
        fswap(*this, // *this <-> saved
             *reinterpret_cast<Union*>(saved));
    }
```
destroy(myTag); // destroy original *this
memcpy(this, saved, sizeof(Union)); // install new *this
}
catch (...) // copy threw
{
    memcpy(this, saved, sizeof(Union)); // roll back: restore *this
    throw;
}

The source distribution contains `yo/containers/examples/unrestricted2.cc` offering a small demo-program in which the here developed `Data` class is used.
Chapter 13

Inheritance

When programming in C, programming problems are commonly approached using a top-down structured approach: functions and actions of the program are defined in terms of sub-functions, which again are defined in sub-sub-functions, etc.. This yields a hierarchy of code: main at the top, followed by a level of functions which are called from main, etc..

In C++ the relationship between code and data is also frequently defined in terms of dependencies among classes. This looks like composition (see section 7.3), where objects of a class contain objects of another class as their data. But the relation described here is of a different kind: a class can be defined in terms of an older, pre-existing, class. This produces a new class having all the functionality of the older class, and additionally defining its own specific functionality. Instead of composition, where a given class contains another class, we here refer to derivation, where a given class is or is-implemented-in-terms-of another class.

Another term for derivation is inheritance: the new class inherits the functionality of an existing class, while the existing class does not appear as a data member in the interface of the new class. When discussing inheritance the existing class is called the base class, while the new class is called the derived class.

Derivation of classes is often used when the methodology of C++ program development is fully exploited. In this chapter we first address the syntactic possibilities offered by C++ for deriving classes. Following this we address some of the specific possibilities offered by class derivation (inheritance).

As we have seen in the introductory chapter (see section 2.4), in the object-oriented approach to problem solving classes are identified during the problem analysis. Under this approach objects of the defined classes represent entities that can be observed in the problem at hand. The classes are placed in a hierarchy, with the top-level class containing limited functionality. Each new derivation (and hence descent in the class hierarchy) adds new functionality compared to yet existing classes.

In this chapter we shall use a simple vehicle classification system to build a hierarchy of classes. The first class is Vehicle, which implements as its functionality the possibility to set or retrieve the mass of a vehicle. The next level in the object hierarchy are land-, water- and air vehicles.

The initial object hierarchy is illustrated in Figure 13.1.

This chapter mainly focuses on the technicalities of class derivation. The distinction between inheritance used to create derived classes whose objects should be considered objects of the base class and inheritance used to implement derived classes in-terms-of their base classes is postponed until the next chapter (14).

Inheritance (and polymorphism, cf. chapter 14) can be used with classes and structs. It is not defined
13.1 Related types

The relationship between the proposed classes representing different kinds of vehicles is further investigated here. The figure shows the object hierarchy: a Car is a special case of a Land vehicle, which in turn is a special case of a Vehicle.

The class Vehicle represents the ‘greatest common divisor’ in the classification system. Vehicle is given limited functionality: it can store and retrieve a vehicle’s mass:

```cpp
class Vehicle
{
    size_t d_mass;

    public:
        Vehicle();
        Vehicle(size_t mass);

        size_t mass() const;
        void setMass(size_t mass);
};
```

Using this class, the vehicle’s mass can be defined as soon as the corresponding object has been created. At a later stage the mass can be changed or retrieved.

To represent vehicles travelling over land, a new class Land can be defined offering Vehicle’s functionality and adding its own specific functionality. Assume we are interested in the speed of land vehicles and in their mass. The relationship between Vehicles and Lands could of course be represented by composition but that would be awkward: composition suggests that a Land vehicle is-implemented-in-terms-of, i.e., contains, a Vehicle, while the natural relationship clearly is that the Land vehicle is a kind of Vehicle.

A relationship in terms of composition would also somewhat complicate our Land class’s design. Consider the following example showing a class Land using composition (only the setMass functionality is shown):
Using composition, the `Land::setMass` function only passes its argument on to `Vehicle::setMass`. Thus, as far as mass handling is concerned, `Land::setMass` introduces no extra functionality, just extra code. Clearly this code duplication is superfluous: a Land object is a Vehicle; to state that a Land object contains a Vehicle is at least somewhat peculiar.

The intended relationship is represented better by inheritance. A rule of thumb for choosing between inheritance and composition distinguishes between *is-a* and *has-a* relationships. A truck *is* a vehicle, so Truck should probably derive from Vehicle. On the other hand, a truck *has* an engine; if you need to model engines in your system, you should probably express this by composing an Engine class inside the Truck class.

Following the above rule of thumb, Land is derived from the base class Vehicle:

```cpp
class Land: public Vehicle
{
    size_t d_speed;
    public:
        Land();
        Land(size_t mass, size_t speed);
        void setspeed(size_t speed);
        size_t speed() const;
};
```

To derive a class (e.g., Land) from another class (e.g., Vehicle) postfix the class name Land in its interface by: public Vehicle:

```cpp
class Land: public Vehicle
{
    size_t d_speed;
    public:
        Land();
        Land(size_t mass, size_t speed);
        void setspeed(size_t speed);
        size_t speed() const;
};
```

The class Land now contains all the functionality of its base class Vehicle as well as its own features. Here those features are a constructor expecting two arguments and member functions to access the d_speed data member. Here is an example showing the possibilities of the derived class Land:

```cpp
Land veh(1200, 145);

int main()
{
    cout << "Vehicle weighs " << veh.mass() << ";\n";
    "its speed is " << veh.speed() << ‘\n’;
}
This example illustrates two features of derivation:

- First, mass is not mentioned as a member in Land's interface. Nevertheless it is used in veh.mass. This member function is an implicit part of the class, inherited from its 'parent' vehicle.

- Second, although the derived class Land contains the functionality of Vehicle, the Vehicle's private members remain private: they can only be accessed by Vehicle's own member functions. This means that Land's member functions must use Vehicle's member functions (like mass and setMass) to address the mass field. Here there's no difference between the access rights granted to Land and the access rights granted to other code outside of the class Vehicle. The class Vehicle encapsulates the specific Vehicle characteristics, and data hiding is one way to realize encapsulation.

Encapsulation is a core principle of good class design. Encapsulation reduces the dependencies among classes improving the maintainability and testability of classes and allowing us to modify classes without the need to modify depending code. By strictly complying with the principle of data hiding a class's internal data organization may change without requiring depending code to be changed as well. E.g., a class Lines originally storing C-strings could at some point have its data organization changed. It could abandon its char ** storage in favor of a vector<string> based storage. When Lines uses perfect data hiding depending source code may use the new Lines class without requiring any modification at all.

As a rule of thumb, derived classes must be fully recompiled (but don't have to be modified) when the data organization (i.e., the data members) of their base classes change. Adding new member functions to the base class doesn't alter the data organization so no recompilation is needed when new member functions are added.

There is one subtle exception to this rule of thumb: if a new member function is added to a base class and that function happens to be declared as the first virtual member function of the base class (cf. chapter 14 for a discussion of the virtual member function concept) then that also changes the data organization of the base class.

Now that Land has been derived from Vehicle we're ready for our next class derivation. We'll define a class Car to represent automobiles. Agreeing that a Car object is a Land vehicle, and that a Car has a brand name it's easy to design the class Car:

```cpp
class Car: public Land
{
    std::string d_brandName;

    public:
    Car();
    Car(size_t mass, size_t speed, std::string const &name);

    std::string const &brandName() const;
};
```

In the above class definition, Car was derived from Land, which in turn is derived from Vehicle. This is called nested derivation: Land is called Car's direct base class, while Vehicle is called Car's indirect base class.
13.2. ACCESS RIGHTS: PUBLIC, PRIVATE, PROTECTED

13.1.1 Inheritance depth: desirable?

Now that Car has been derived from Land and Land has been derived from Vehicle we might easily be seduced into thinking that these class hierarchies are the way to go when designing classes. But maybe we should temper our enthusiasm.

Repeatedly deriving classes from classes quickly results in big, complex class hierarchies that are hard to understand, hard to use and hard to maintain. Hard to understand and use as users of our derived class now also have to learn all its (indirect) base class features as well. Hard to maintain because all those classes are very closely coupled. While it may be true that when data hiding is meticulously adhered to derived classes do not have to be modified when their base classes alter their data organization, it also quickly becomes practically infeasible to change those base classes once more and more (derived) classes depend on their current organization.

What initially looks like a big gain, inheriting the base class’s interface, thus becomes a liability. The base class’s interface is hardly ever completely required and in the end a class may benefit from explicitly defining its own member functions rather than obtaining them through inheritance.

Often classes can be defined in-terms-of existing classes: some of their features are used, but others need to be shielded off. Consider the stack container: it is commonly implemented in-terms-of a deque, returning deque::back’s value as stack::top’s value.

When using inheritance to implement an is-a relationship make sure to get the ‘direction of use’ right: inheritance aiming at implementing an is-a relationship should focus on the base class: the base class facilities aren’t there to be used by the derived class, but the derived class facilities should redefine (reimplement) the base class facilities using polymorphism (which is the topic of the next chapter), allowing code to use the derived class facilities polymorphically through the base class.

We’ve seen this approach when studying streams: the base class (e.g., ostream) is used time and again. The facilities defined by classes derived from ostream (like ofstream and ostringstream) are then used by code only relying on the facilities offered by the ostream class, never using the derived classes directly.

When designing classes always aim at the lowest possible coupling. Big class hierarchies usually indicate poor understanding of robust class design. When a class’s interface is only partially used and if the derived class is implemented in terms of another class consider using composition rather than inheritance and define the appropriate interface members in terms of the members offered by the composed objects.

13.2 Access rights: public, private, protected

Early in the C++ Annotations (cf. section 3.2.1) we encountered two important design principles when developing classes: data hiding and encapsulation. Data hiding restricts control over an object’s data to the members of its class, encapsulating is used to restrict access to the functionality of objects. Both principles are invaluable tools for maintaining data integrity.

The keyword private starts sections in class interfaces in which members are declared which can only be accessed by members of the class itself. This is our main tool for realizing data hiding. According to established good practices of class design the public sections are populated with member functions offering a clean interface to the class’s functionality. These members allow users to communicate with objects; leaving it to the objects how requests sent to objects are handled. In a well-designed class its objects are in full control of their data.

Inheritance doesn’t change these principles, nor does it change the way the private and protected keywords operate. A derived class does not have access to a base class’s private section.
Sometimes this is a bit too restrictive. Consider a class implementing a random number generating `streambuf` (cf. chapter 6). Such a `streambuf` can be used to construct an `istream irand`, after which extractions from `irand` produces the next random number, like in the next example in which 10 random numbers are generated using stream I/O:

```c++
RandBuf buffer;
istream irand(&buffer);

for (size_t idx = 0; idx != 10; ++idx)
{
    size_t next;
    irand >> next;
    cout << "next random number: " << next << 'n';
}
```

The question is, how many random numbers should `irand` be able to generate? Fortunately, there's no need to answer this question, as `RandBuf` can be made responsible for generating the next random number. `RandBuf`, therefore, operates as follows:

- It generates a random number;
- It is passed in textual form to its base class `streambuf`;
- The `istream object` extracts this random number, merely using `streambuf`'s interface;
- this process is repeated for subsequent random numbers;

Once `RandBuf` has stored the text representation of the next random number in some buffer, it must tell its base class (`streambuf`) where to find the random number's characters. For this `streambuf` offers a member `setg`, expecting the location and size of the buffer holding the random number's characters.

The member `setg` clearly cannot be declared in `streambuf`'s private section, as `RandBuf` must use it to prepare for the extraction of the next random number. But it should also not be in `streambuf`'s public section, as that could easily result in unexpected behavior by `irand`. Consider the following hypothetical example:

```c++
RandBuf buffer;
istream irand(&buffer);

char buffer[] = "12";
buffer.setg(buffer, ...); // setg public: buffer now contains 12

size_t next;
irand >> next; // not a *random* value, but 12.
```

Clearly there is a close connection between `streambuf` and its derived class `RandBuf`. By allowing `RandBuf` to specify the buffer from which `streambuf` reads characters `RandBuf` remains in control, denying other parts of the program to break its well-defined behavior.

This close connection between base- and derived-classes is realized by a third keyword related to the accessibility of class members: `protected`. Here is how the member `setg` could have been be declared in a class `streambuf`:

```c++
class streambuf
```
13.2. ACCESS RIGHTS: PUBLIC, PRIVATE, PROTECTED

protected:
    void setg(... parameters ...); // available to derived classes

public:
    // public members here

Protected members are members that can be accessed by derived classes, but are not part of a class's public interface.

Avoid the temptation to declare data members in a class's protected section: it's a sure sign of bad class design as it needlessly results in tight coupling of base and derived classes. The principle of data hiding should not be abandoned now that the keyword protected has been introduced. If a derived class (but not other parts of the software) should be given access to its base class's data, use member functions: accessors and modifiers declared in the base class's protected section. This enforces the intended restricted access without resulting in tightly coupled classes.

13.2.1 Public, protected and private derivation

With inheritance public derivation is frequently used. When public derivation is used the access rights of the base class's interface remains unaltered in the derived class. But the type of inheritance may also be defined as private or protected.

Protected derivation is used when the keyword protected is put in front of the derived class's base class:

    class Derived: protected Base

When protected derivation is used all the base class's public and protected members become protected members in the derived class. The derived class may access all the base class's public and protected members. Classes that are in turn derived from the derived class view the base class's members as protected. Any other code (outside of the inheritance tree) is unable to access the base class's members.

Private derivation is used when the keyword private is put in front of the derived class's base class:

    class Derived: private Base

When private derivation is used all the base class's members turn into private members in the derived class. The derived class members may access all base class public and protected members but base class members cannot be used elsewhere.

Public derivation should be used to define an is-a relationship between a derived class and a base class: the derived class object is-a base class object allowing the derived class object to be used polymorphically as a base class object in code expecting a base class object. Private inheritance is used in situations where a derived class object is defined in-terms-of the base class where composition cannot be used. There's little documented use for protected inheritance, but one could maybe encounter protected inheritance when defining a base class that is itself a derived class and needs to make its base class members available to classes derived from itself.
Combinations of inheritance types do occur. For example, when designing a stream-class it is usually derived from `std::istream` or `std::ostream`. However, before a stream can be constructed, a `std::streambuf` must be available. Taking advantage of the fact that the inheritance order is defined in the class interface, we use multiple inheritance (see section 13.7) to derive the class from both `std::streambuf` and (then) from `std::ostream`. To the class's users it is a `std::ostream` and not a `std::streambuf`. So private derivation is used for the latter, and public derivation for the former class:

```cpp
class Derived: private std::streambuf, public std::ostream
```

### 13.2.2 Promoting access rights

When private or protected derivation is used, users of derived class objects are denied access to the base class members. Private derivation denies access to all base class members to users of the derived class, protected derivation does the same, but allows classes that are in turn derived from the derived class to access the base class's public and protected members.

In some situations this scheme is too restrictive. Consider a class `RandStream` derived privately from a class `RandBuf` which is itself derived from `std::streambuf` and also publicly from `istream`:

```cpp
class RandBuf: public std::streambuf
{
  // implements a buffer for random numbers
};
class RandStream: private RandBuf, public std::istream
{
  // implements a stream to extract random values from
};
```

Such a class could be used to extract, e.g., random numbers using the standard `istream` interface.

Although the `RandStream` class is constructed with the functionality of `istream` objects in mind, some of the members of the class `std::streambuf` may be considered useful by themselves. E.g., the function `std::streambuf::in_avail` returns a lower bound to the number of characters that can be read immediately. The standard way to make this function available is to define a *shadow member* calling the base class's member:

```cpp
class RandStream: private RandBuf, public std::istream
{
  // implements a stream to extract random values from
  public:
    std::streamsize in_avail();
};
inline std::streamsize RandStream::in_avail()
{
  return std::streambuf::in_avail();
}
```

This looks like a lot of work for just making available a member from the protected or private base classes. If the intent is to make available the `in_avail` member access promotion can be used. Access promotion allows us to specify which members of private (or protected) base classes become
available in the protected (or public) interface of the derived class. Here is the above example, now using access promotion:

```cpp
class RandStream: private RandBuf, public std::istream
{
  // implements a stream to extract random values from
  public:
    using std::streambuf::in_avail;
};
```

It should be noted that access promotion makes available all overloaded versions of the declared base class member. So, if `std::streambuf` would offer not only `in_avail` but also, e.g., `in_avail(size_t *)` both members would become part of the public interface.

### 13.3 The constructor of a derived class

A derived class inherits functionality from its base class (or base classes, as C++ supports multiple inheritance, cf. section 13.7). When a derived class object is constructed it is built on top of its base class object. As a consequence the base class must have been constructed before the actual derived class elements can be initialized. This results in some requirements that must be observed when defining derived class constructors.

A constructor exists to initialize the object’s data members. A derived class constructor is also responsible for the proper initialization of its base class. Looking at the definition of the class `Land` introduced earlier (section 13.1), its constructor could simply be defined as follows:

```cpp
Land::Land(size_t mass, size_t speed)
{
  setMass(mass);
  setspeed(speed);
}
```

However, this implementation has several disadvantages.

- When constructing a derived class object a base class constructor is always called before any action is performed on the derived class object itself. By default the base class’s default constructor is going to be called.

- Using the base class constructor only to reassign new values to its data members in the derived class constructor’s body usually is inefficient, but sometimes sheer impossible as in situations where base class reference or const data members must be initialized. In those cases a specialized base class constructor must be used instead of the base class default constructor.

A derived class’s base class may be initialized using a dedicated base class constructor by calling the base class constructor in the derived class constructor’s initializer clause. Calling a base class constructor in a constructor’s initializer clause is called a **base class initializer**. The base class initializer must be called before initializing any of the derived class’s data members and when using the base class initializer none of the derived class data members may be used. When constructing a derived class object the base class is constructed first and only after that construction has successfully completed the derived class data members are available for initialization. `Land`’s constructor may therefore be improved:
Land::Land(size_t mass, size_t speed) : Vehicle(mass), d_speed(speed) {} 

Derived class constructors always by default call their base class's default constructor. This is of course not correct for a derived class's copy constructor. Assuming that the class Land must be provided with a copy constructor it may use the Land const &other to represent the other's base class:

Land::Land(Land const &other) // assume a copy constructor is needed :
    Vehicle(other), // copy-construct the base class part.
    d_speed(other.speed) // copy-construct Land’s data members
{}

13.3.1 Move construction

As with classes using composition derived classes may benefit from defining a move constructor. A derived class may offer a move constructor for two reasons:

- it supports move construction for its data members
- its base class is move-aware

The design of move constructors moving data members was covered in section 9.7. A move constructor for a derived class whose base class is move-aware must anonymize the rvalue reference before passing it to the base class move constructor. The std::move function should be used when implementing the move constructor to move the information in base classes or composed objects to their new destination object.

The first example shows the move constructor for the class Car, assuming it has a movable char *d_brandName data member and assuming that Land is a move-aware class. The second example shows the move constructor for the class Land, assuming that it does not itself have movable data members, but that its Vehicle base class is move-aware:

Car::Car(Car &&tmp) :
    Land(std::move(tmp)), // anonimize 'tmp'
    d_brandName(tmp.d_brandName) // move the char *’s value
{} 

Land(Land &&tmp) :
    Vehicle(std::move(tmp)), // move-aware Vehicle
    d_speed(tmp.d_speed) // plain copying of plain data
{}
13.3.2 Move assignment

Derived classes may also benefit from move assignment operations. If the derived class and its base class support swapping then the implementation is simple, following the standard shown earlier in section 9.7.3. For the class Car this could boil down to:

```cpp
Car &Car::operator=(Car &&tmp)
{
    swap(tmp);
    return *this;
}
```

If swapping is not supported then `std::move` can be used to call the base class’s move assignment operator:

```cpp
Car &Car::operator=(Car &&tmp)
{
    static_cast<Land &>(*this) = std::move(tmp);
    // move Car’s own data members next
    return *this;
}
```

13.3.3 Inheriting constructors

The C++11 standard allows derived classes to be constructed without explicitly defining derived class constructors. In those cases the available base class constructors are called.

This feature is either used or not. It is not possible to omit some of the derived class constructors, using the corresponding base class constructors instead. To use this feature for classes that are derived from multiple base classes (cf. section 13.7) all the base class constructors must have different signatures. Considering the complexities that are involved here it’s probably best to avoid using base class constructors for classes using multiple inheritance.

The construction of derived class objects can be delegated to base class constructor(s) using the following syntax:

```cpp
class BaseClass
{
    public:
        // BaseClass constructor(s)
};

class DerivedClass: public BaseClass
{
    public:
        using BaseClass::BaseClass; // No DerivedClass constructors
        // are defined
};
```
13.4 The destructor of a derived class

Destructors of classes are automatically called when an object is destroyed. This also holds true for objects of classes derived from other classes. Assume we have the following situation:

```cpp
class Base
{
    public:
        ~Base();
};

class Derived: public Base
{
    public:
        ~Derived();
};

int main()
{
    Derived derived;
}
```

At the end of `main`, the `derived` object ceases to exist. Hence, its destructor (`~Derived`) is called. However, since `derived` is also a `Base` object, the `~Base` destructor is called as well. The base class destructor is never explicitly called from the derived class destructor.

Constructors and destructors are called in a stack-like fashion: when `derived` is constructed, the appropriate base class constructor is called first, then the appropriate derived class constructor is called. When the object `derived` is destroyed, its destructor is called first, automatically followed by the activation of the `Base` class destructor. A derived class destructor is always called before its base class destructor is called.

When the construction of a derived class objects did not successfully complete (i.e., the constructor threw an exception) then its destructor is not called. However, the destructors of properly constructed base classes will be called if a derived class constructor throws an exception. This, of course, is it should be: a properly constructed object should also be destroyed, eventually. Example:

```cpp
#include <iostream>
struct Base
{
    ~Base()
    {
        std::cout << "Base destructor\n";
    }
};
struct Derived: public Base
{
    Derived()
    {
        throw 1; // at this time Base has been constructed
    }
};
int main()
{
```
try
{
    Derived d;
} catch(...) { } /* This program displays 'Base destructor' */

13.5 Redefining member functions

Derived classes may redefine base class members. Let’s assume that a vehicle classification system must also cover trucks, consisting of two parts: the front part, the tractor, pulls the rear part, the trailer. Both the tractor and the trailer have their own mass, and the mass function should return the combined mass.

The definition of a Truck starts with a class definition. Our initial Truck class is derived from Car but it is then expanded to hold one more size_t field representing the additional mass information. Here we choose to represent the mass of the tractor in the Car class and to store the mass of full truck (tractor + trailer) in its own d_mass data member:

```cpp
class Truck: public Car
{
    size_t d_mass;

    public:
    Truck();
    Truck(size_t tractor_mass, size_t speed, char const *name, size_t trailer_mass);

    void setMass(size_t tractor_mass, size_t trailer_mass);
    size_t mass() const;
};
```

```cpp
Truck::Truck(size_t tractor_mass, size_t speed, char const *name, size_t trailer_mass) :
    Car(tractor_mass, speed, name) // Car::setMass and mass.
{
    d_mass = trailer_mass + trailer_mass;
}
```

Note that the class Truck now contains two functions already present in the base class Car: `setMass` and `mass`.

- The redefinition of `setMass` poses no problems: this function is simply redefined to perform actions which are specific to a Truck object.
- Redefining `setMass`, however, hides `Car::setMass`. For a Truck only the `setMass` function having two size_t arguments can be used.
• The Vehicle’s setMass function remains available for a Truck, but it must now be called explicitly, as Car::setMass is hidden from view. This latter function is hidden, even though Car::setMass has only one size_t argument. To implement Truck::setMass we could write:

```cpp
void Truck::setMass(size_t tractor_mass, size_t trailer_mass) {
    d_mass = tractor_mass + trailer_mass;
    Car::setMass(tractor_mass); // note: Car:: is required
}
```

• Outside of the class Car::setMass is accessed using the scope resolution operator. So, if a Truck truck needs to set its Car mass, it must use

```cpp
truck.Car::setMass(x);
```

• An alternative to using the scope resolution operator is to add a member having the same function prototype as the base class member to the derived class's interface. This derived class member could be implemented inline to call the base class member. E.g., we add the following member to the class Truck:

```cpp
// in the interface:
void setMass(size_t tractor_mass);

// below the interface:
inline void Truck::setMass(size_t tractor_mass) {
    d_mass -= Car::mass() + tractor_mass;
    Car::setMass(tractor_mass);
}
```

Now the single argument setMass member function can be used by Truck objects without using the scope resolution operator. As the function is defined inline, no overhead of an additional function call is involved.

• To prevent hiding the base class members a using declaration may be added to the derived class interface. The relevant section of Truck’s class interface then becomes:

```cpp
class Truck: public Car {
    public:
        using Car::setMass;
        void setMass(size_t tractor_mass, size_t trailer_mass);
};
```

A using declaration imports (all overloaded versions of) the mentioned member function directly into the derived class’s interface. If a base class member has a signature that is identical to a derived class member then compilation fails (a using Car::mass declaration cannot be added to Truck’s interface). Now code may use truck.setMass(5000) as well as truck.setMass(5000, 2000).

Using declarations obey access rights. To prevent non-class members from using setMass(5000) without a scope resolution operator but allowing derived class members to do so the using Car::setMass declaration should be put in the class Truck’s private section.
The function `mass` is also already defined in `Car`, as it was inherited from `Vehicle`. In this case, the class `Truck` redefines this member function to return the truck's full mass:

```cpp
size_t Truck::mass() const
{
    return d_mass;
}
```

Example:

```cpp
int main()
{
    Land vehicle(1200, 145);
    Truck lorry(3000, 120, "Juggernaut", 2500);
    lorry.Vehicle::setMass(4000);
    cout << '\n' << "Tractor weighs " <<
        lorry.Vehicle::mass() << '\n' <<
        "Truck + trailer weighs " << lorry.mass() << '\n' <<
        "Speed is " << lorry.speed() << '\n' <<
        "Name is " << lorry.name() << '\n';
}
```

The class `Truck` was derived from `Car`. However, one might question this class design. Since a truck is conceived of as a combination of an tractor and a trailer it is probably better defined using a mixed design, using inheritance for the tractor part (inheriting from `Car`, and composition for the trailer part).

This redesign changes our point of view from a `Truck` being a `Car` (and some strangely added data members) to a `Truck` still being an `Car` (the tractor) and containing a `Vehicle` (the trailer).

`Truck`'s interface is now very specific, not requiring users to study `Car`'s and `Vehicle`'s interfaces and it opens up possibilities for defining 'road trains': tractors towing multiple trailers. Here is an example of such an alternate class setup:

```cpp
class Truck: public Car // the tractor
{
    Vehicle d_trailer; // use vector<Vehicle> for road trains
    public:
        Truck();
        Truck(size_t tractor_mass, size_t speed, char const *name,
              size_t trailer_mass);
        void setMass(size_t tractor_mass, size_t trailer_mass);
        void setTractorMass(size_t tractor_mass);
        void setTrailerMass(size_t trailer_mass);
        size_t tractorMass() const;
        size_t trailerMass() const;
        // consider:
        Vehicle const &trailer() const;
};
```
13.6 i/ostream::init

Consider classes derived from `std::istream` or `std::ostream`. Such a class could be designed as follows:

```cpp
class XIstream: public std::istream
{
    public:
        ...
};
```

Assuming that the streambuf to which XIstream interfaces is not yet available construction time, XIstream only offers default constructors. The class could, however, offer a member `void switchStream(std::streambuf *sb)` to provide XIstream objects with a streambuf to interface to. How to implement switchStream?

The classes `std::istream` and `std::ostream` offer a protected member `void init(std::streambuf *sb)` to realize this. The init member expects a pointer to a streambuf which is associated with the `istream` or `ostream` object. The init member properly ends any existing association before switching to the streambuf whose address is provided to init.

Assuming that the streambuf to which switchStream's sb points persists, then switchStream could simply be implemented like this:

```cpp
void switchStream(streambuf *sb)
{
    init(sb);
}
```

13.7 Multiple inheritance

Up to now, a class has always been derived from a single base class. In addition to single inheritance C++ also supports multiple inheritance. In multiple inheritance a class is derived from several base classes and hence inherits functionality from multiple parent classes at the same time.

When using multiple inheritance it should be defensible to consider the newly derived class an instantiation of both base classes. Otherwise, composition is more appropriate. In general, linear derivation (using only one base class) is used much more frequently than multiple derivation. Good class design dictates that a class should have a single, well described responsibility and that principle often conflicts with multiple inheritance where we can state that objects of class Derived are both Base1 and Base2 objects.

But then, consider the prototype of an object for which multiple inheritance was used to its extreme: the Swiss army knife! This object is a knife, it is a pair of scissors, it is a can-opener, it is a corkscrew, it is ....

The ‘Swiss army knife’ is an extreme example of multiple inheritance. In C++ there are some good reasons, not violating the ‘one class, one responsibility’ principle that is covered in the next chapter. In this section the technical details of constructing classes using multiple inheritance are discussed.

How to construct a ‘Swiss army knife’ in C++? First we need (at least) two base classes. For example, let’s assume we are designing a toolkit allowing us to construct an instrument panel of an aircraft’s cockpit. We design all kinds of instruments, like an artificial horizon and an altimeter. One of the
components that is often seen in aircraft is a nav-com set: a combination of a navigational beacon receiver (the ‘nav’ part) and a radio communication unit (the ‘com’-part). To define the nav-com set, we start by designing the NavSet class (assume the existence of the classes Intercom, VHF.Dial and Message):

    class NavSet
    {
        public:
            NavSet(Intercom &intercom, VHF.Dial &dial);
            size_t activeFrequency() const;
            size_t standByFrequency() const;
            void setStandByFrequency(size_t freq);
            size_t toggleActiveStandby();
            void setVolume(size_t level);
            void identEmphasis(bool on_off);
    };

Next we design the class ComSet:

    class ComSet
    {
        public:
            ComSet(Intercom &intercom);
            size_t frequency() const;
            size_t passiveFrequency() const;
            void setPassiveFrequency(size_t freq);
            size_t toggleFrequencies();
            void setAudioLevel(size_t level);
            void powerOn(bool on_off);
            void testState(bool on_off);
            void transmit(Message &message);
    };

Using objects of this class we can receive messages, transmitted though the Intercom, but we can also transmit messages using a Message object that’s passed to the ComSet object using its transmit member function.

Now we’re ready to construct our NavCom set:

    class NavComSet: public ComSet, public NavSet
    {
        public:
            NavComSet(Intercom &intercom, VHF.Dial &dial);
    };

Done. Now we have defined a NavComSet which is both a NavSet and a ComSet: the facilities of both base classes are now available in the derived class using multiple inheritance.
Please note the following:

- The keyword `public` is present before both base class names (`NavSet` and `ComSet`). By default, inheritance uses private derivation and the keyword `public` must be repeated before each of the base class specifications. Base classes are not required to use the same derivation type. One base class could have public derivation and another base class could use private derivation.

- The multiply derived class `NavComSet` introduces no additional functionality of its own, but merely combines two existing classes into a new aggregate class. Thus, C++ offers the possibility to simply sweep multiple simple classes into one more complex class.

- Here is the implementation of the `NavComSet` constructor:

```cpp
NavComSet::NavComSet(Intercom &intercom, VHF_Dial &dial) :
    ComSet(intercom),
    NavSet(intercom, dial)
{};
```

The constructor requires no extra code: Its purpose is to activate the constructors of its base classes. The order in which the base class initializers are called is not dictated by their calling order in the constructor's code, but by the ordering of the base classes in the class interface.

- The `NavComSet` class definition requires no additional data members or member functions: here (and often) the inherited interfaces provide all the required functionality and data for the multiply derived class to operate properly.

Of course, while defining the base classes, we made life easy on ourselves by strictly using different member function names. So, there is a function `setVolume` in the `NavSet` class and a function `setAudioLevel` in the `ComSet` class. A bit cheating, since we could expect that both units in fact have a composed object `Amplifier`, handling the volume setting. A revised class might offer an `Amplifier &amplifier() const` member function, and leave it to the application to set up its own interface to the amplifier. Alternatively, a revised class could define members for setting the volume of either the `NavSet` or the `ComSet` parts.

In situations where two base classes offer identically named members special provisions need to be made to prevent ambiguity:

- The intended base class can explicitly be specified using the base class name and scope resolution operator:

```cpp
NavComSet navcom(intercom, dial);
```

```cpp
navcom.NavSet::setVolume(5);  // sets the NavSet volume level
navcom.ComSet::setVolume(5);  // sets the ComSet volume level
```

- The class interface is provided with member functions that can be called unambiguously. These additional members are usually defined inline:

```cpp
class NavComSet: public ComSet, public NavSet
{
    public:
        NavComSet(Intercom &intercom, VHF_Dial &dial);
        void comVolume(size_t volume);
        void navVolume(size_t volume);
};
```
13.8 Conversions between base classes and derived classes

When public inheritance is used to define classes, an object of a derived class is at the same time an object of the base class. This has important consequences for object assignment and for the situation where pointers or references to such objects are used. Both situations are now discussed.

13.8.1 Conversions with object assignments

Continuing our discussion of the `NavCom` class, introduced in section 13.7, we now define two objects, a base class and a derived class object:

```cpp
ComSet com(intercom);
NavComSet navcom(intercom2, dial2);
```

The object `navcom` is constructed using an `Intercom` and a `VHF_Dial` object. However, a `NavComSet` is at the same time a `ComSet`, allowing the assignment from `navcom` (a derived class object) to `com` (a base class object):

```cpp
com = navcom;
```
The effect of this assignment is that the object com now communicates with intercom2. As a ComSet does not have a VHF_Dial, the navcom's dial is ignored by the assignment. When assigning a base class object from a derived class object only the base class data members are assigned, other data members are dropped, a phenomenon called slicing. In situations like these slicing probably does not have serious consequences, but when passing derived class objects to functions defining base class parameters or when returning derived class objects from functions returning base class objects slicing also occurs and might have unwelcome side-effects.

The assignment from a base class object to a derived class object is problematic. In a statement like

```
navcom = com;
```

it isn’t clear how to reassign the NavComSet’s VHF_Dial data member as they are missing in the ComSet object com. Such an assignment is therefore refused by the compiler. Although derived class objects are also base class objects, the reverse does not hold true: a base class object is not also a derived class object.

The following general rule applies: in assignments in which base class objects and derived class objects are involved, assignments in which data are dropped are legal (called slicing). Assignments in which data remain unspecified are not allowed. Of course, it is possible to overload an assignment operator to allow the assignment of a derived class object from a base class object. To compile the statement

```
navcom = com;
```

the class NavComSet must have defined an overloaded assignment operator accepting a ComSet object for its argument. In that case it’s up to the programmer to decide what the assignment operator will do with the missing data.

### 13.8.2 Conversions with pointer assignments

We return to our Vehicle classes, and define the following objects and pointer variable:

```c
Land land(1200, 130);
Car auto(500, 75, "Daf");
Truck truck(2600, 120, "Mercedes", 6000);
Vehicle *vp;
```

Now we can assign the addresses of the three objects of the derived classes to the Vehicle pointer:

```c
vp = &land;
vp = &auto;
vp = &truck;
```

Each of these assignments is acceptable. However, an implicit conversion of the derived class to the base class Vehicle is used, since vp is defined as a pointer to a Vehicle. Hence, when using vp only the member functions manipulating mass can be called as this is the Vehicle’s only functionality. As far as the compiler can tell this is the object vp points to.

The same holds true for references to Vehicles. If, e.g., a function is defined having a Vehicle reference parameter, the function may be passed an object of a class derived from Vehicle. Inside
the function, the specific Vehicle members remain accessible. This analogy between pointers and references holds true in general. Remember that a reference is nothing but a pointer in disguise: it mimics a plain variable, but actually it is a pointer.

This restricted functionality has an important consequence for the class Truck. Following vp = &truck, vp points to a Truck object. So, vp->mass() returns 2600 instead of 8600 (the combined mass of the cabin and of the trailer: 2600 + 6000), which would have been returned by truck.mass().

When a function is called using a pointer to an object, then the type of the pointer (and not the type of the object) determines which member functions are available and can be executed. In other words, C++ implicitly converts the type of an object reached through a pointer to the pointer's type.

If the actual type of the object pointed to by a pointer is known, an explicit type cast can be used to access the full set of member functions that are available for the object:

```cpp
Truck truck;
Vehicle *vp;
vp = &truck; // vp now points to a truck object

Truck *trp;
trp = static_cast<Truck *>(vp);
cout << "Make: " << trp->name() << '\n';
```

Here, the second to last statement specifically casts a Vehicle * variable to a Truck *. As usual (when using casts), this code is not without risk. It only works if vp really points to a Truck. Otherwise the program may produce unexpected results.

### 13.9 Using non-default constructors with new[]

An often heard complaint is that operator new[] calls the default constructor of a class to initialize the allocated objects. For example, to allocate an array of 10 strings we can do

```cpp
new string[10];
```

but it is not possible to use another constructor. Assuming that we'd want to initialize the strings with the text hello world, we can't write something like:

```cpp
new string("hello world")[10];
```

The initialization of a dynamically allocated object usually consists of a two-step process: first the array is allocated (implicitly calling the default constructor); second the array's elements are initialized, as in the following little example:

```cpp
string *sp = new string[10];
fill(sp, sp + 10, string("hello world"));
```

These approaches all suffer from 'double initializations', comparable to not using member initializers in constructors.
One way to avoid double initialization is to use inheritance. Inheritance can profitably be used to call non-default constructors in combination with operator new[]. The approach capitalizes on the following:

- A base class pointer may point to a derived class object;
- A derived class without (non-static) data members has the same size as its base class.

The above also suggests a possible approach:

- Derive a simple, member-less class from the class we’re interested in;
- Use the appropriate base class initializer in its default constructor;
- Allocate the required number of derived class objects, and assign new[]’s return expression to a pointer to base class objects.

Here is a simple example, producing 10 lines containing the text **hello world**:

```cpp
#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;

struct Xstr: public string
{
    Xstr()
    :
        string("hello world")
    {}
};

int main()
{
    string *sp = new Xstr[10];
    copy(sp, sp + 10, ostream_iterator<string>(cout, "\n"));
}
```

Of course, the above example is fairly unsophisticated, but it’s easy to polish the example: the class `Xstr` can be defined in an anonymous namespace, accessible only to a function `getString()` which may be given a `size_t nObjects` parameter, allowing users to specify the number of `hello world`-initialized strings they would like to allocate.

Instead of hard-coding the base class arguments it’s also possible to use variables or functions providing the appropriate values for the base class constructor’s arguments. In the next example a **local class** `Xstr` is defined inside a function `nStrings(size_t nObjects, char const *fname)`, expecting the number of `string` objects to allocate and the name of a file whose subsequent lines are used to initialize the objects. The local class is invisible outside of the function `nStrings`, so no special namespace safeguards are required.

As discussed in section 7.10, members of local classes cannot access local variables from their surrounding function. However, they can access global and static data defined by the surrounding function.
Using a local class neatly allows us to hide the implementation details within the function `nStrings`, which simply opens the file, allocates the objects, and closes the file again. Since the local class is derived from `string`, it can use any `string` constructor for its base class initializer. In this particular case it calls the `string(char const *)` constructor, providing it with subsequent lines of the just opened stream via its static member function `nextLine()`. This latter function is, as it is a static member function, available to `Xstr` default constructor's member initializers even though no `Xstr` object is available by that time.

```cpp
#include <fstream>
#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>

using namespace std;

string *nStrings(size_t size, char const *fname)
{
    static ifstream in;

    struct Xstr: public string
    {
        Xstr()
        :
            string(nextLine())
        {}
        static char const *nextLine()
        {
            static string line;
            getline(in, line);
            return line.c_str();
        }
    };
    in.open(fname);
    string *sp = new Xstr[size];
    in.close();

    return sp;
}

int main()
{
    string *sp = nStrings(10, "nstrings.cc");
    copy(sp, sp + 10, ostream_iterator<string>(cout, "\n"));
}
```

When this program is run, it displays the first 10 lines of the file `nstrings.cc`.

Note that the above implementation can’t safely be used in a multithreaded environment. In that case a `mutex` should be used to protect the three statements just before the function’s return statement.

A completely different way to avoid the double initialization (not using inheritance) is to use placement new (cf. section 9.1.5): simply allocate the required amount of memory followed by the proper
in-place allocation of the objects, using the appropriate constructors. The following example can also be used in multithreaded environments. The approach uses a pair of static construct/destroy members to perform the required initialization.

In the program shown below construct expects a istream that provides the initialization strings for objects of a class String simply containing a std::string object. Construct first allocates enough memory for the n String objects plus room for an initial size_t value. This initial size_t value is then initialized with n. Next, in a for statement, lines are read from the provided stream and the lines are passed to the constructors, using placement new calls. Finally the address of the first String object is returned.

The member destroy handles the destruction of the objects. It retrieves the number of objects to destroy from the size_t it finds just before the location of the address of the first object to destroy. The objects are then destroyed by explicitly calling their destructors. Finally the raw memory, originally allocated by construct is returned.

```cpp
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

class String
{
    union Ptrs
    {
        void *vp;
        String *sp;
        size_t *np;
    };

    std::string d_str;

public:
    String(std::string const &txt)
    :
        d_str(txt)
    {}

    ~String()
    {
        cout << "destructor: " << d_str << '\n';
    }

    static String *construct(istream &in, size_t n)
    {
        Ptrs p = {operator new(n * sizeof(String) + sizeof(size_t))};
        *p.np++ = n;

        string line;
        for (size_t idx = 0; idx != n; ++idx)
        {
            getline(in, line);
            new(p.sp + idx) String(line);
        }

        return p.sp;
    }
};
```
static void destroy(String *sp)
{
    Ptrs p = {sp};
    --p.np;
    for (size_t n = *p.np; n--; )
        sp++->~String();

    operator delete (p.vp);
}

int main()
{
    String *sp = String::construct(cin, 5);

    String::destroy(sp);
}

/*
After providing 5 lines containing, respectively
    alfa, bravo, charlie, delta, echo
the program displays:
    destructor: alfa
destructor: bravo
destructor: charlie
destructor: delta
destructor: echo
*/
Chapter 14

Polymorphism

Using inheritance classes may be derived from other classes, called base classes. In the previous chapter we saw that base class pointers may be used to point to derived class objects. We also saw that when a base class pointer points to an object of a derived class it is the type of the pointer rather than the type of the object it points to what determines which member functions are visible. So when a `Vehicle *vp` points to a `Car` object `Car`'s speed or `brandName` members can’t be used.

In the previous chapter two fundamental ways classes may be related to each other were discussed: a class may be `implemented-in-terms-of` another class and it can be stated that a derived class is-a base class. The former relationship is usually implemented using composition, the latter is usually implemented using a special form of inheritance, called polymorphism, the topic of this chapter.

An is-a relationship between classes allows us to apply the Liskov Substitution Principle (LSP) according to which a derived class object may be passed to and used by code expecting a pointer or reference to a base class object. In the C++ Annotations so far the LSP has been applied many times. Every time an `ostringstream, ofstream` or `fstream` was passed to functions expecting an `ostream` we’ve been applying this principle. In this chapter we’ll discover how to design our own classes accordingly.

LSP is implemented using a technique called polymorphism: although a base class pointer is used it performs actions defined in the (derived) class of the object it actually points to. So, a `Vehicle *vp` might behave like a `Car *` when pointing to a `Car`.

Polymorphism is implemented using a feature called late binding. It's called that way because the decision which function to call (a base class function or a function of a derived class) cannot be made at compile-time, but is postponed until the program is actually executed: only then it is determined which member function will actually be called.

In C++ late binding is not the default way functions are called. By default static binding (or early binding) is used. With static binding the functions that are called are determined by the compiler, merely using the class types of objects, object pointers or object references.

Late binding is an inherently different (and slightly slower) process as it is decided at run-time, rather than at compile-time what function is going to be called. As C++ supports both late- and early-binding C++ programmers are offered an option as to what kind of binding to use. Choices can be optimized to the situations at hand. Many other languages offering object oriented facilities (e.g., Java) only or by default offer late binding. C++ programmers should be keenly aware of this. Expecting early binding and getting late binding may easily produce nasty bugs.

---

1In one of the StarTrek movies, Capt. Kirk was in trouble, as usual. He met an extremely beautiful lady who, however, later on changed into a hideous troll. Kirk was quite surprised, but the lady told him: “Didn’t you know I am a polymorph?”
Let's look at a simple example to start appreciating the differences between late and early binding. The example merely illustrates. Explanations of why things are as shown are shortly provided.

Consider the following little program:

```cpp
#include <iostream>
using namespace std;

class Base
{
    protected:
        void hello()
        {
            cout << "base hello\n";
        }
    public:
        void process()
        {
            hello();
        }
};
class Derived: public Base
{
    protected:
        void hello()
        {
            cout << "derived hello\n";
        }
};

int main()
{
    Derived derived;
    derived.process();
}
```

The important characteristic of the above program is the `Base::process` function, calling `hello`. As `process` is the only member that is defined in the public interface it is the only member that can be called by code not belonging to the two classes. The class `Derived`, derived from `Base` clearly inherits `Base`'s interface and so `process` is also available in `Derived`. So the `Derived` object in `main` is able to call `process`, but not `hello`.

So far, so good. Nothing new, all this was covered in the previous chapter. One may wonder why `Derived` was defined at all. It was presumably defined to create an implementation of `hello` that's appropriate for `Derived` but differing from `Base::hello`'s implementation. `Derived`'s author's reasoning was as follows: `Base`'s implementation of `hello` is not appropriate; a `Derived` class object can remedy that by providing an appropriate implementation. Furthermore our author reasoned:

"since the type of an object determines the interface that is used, process must call Derived::hello as hello is called via process from a Derived class object".

Unfortunately our author's reasoning is flawed, due to static binding. When `Base::process` was compiled static binding caused the compiler to bind the hello call to `Base::hello()`.
The author intended to create a Derived class that is-a Base class. That only partially succeeded: Base's interface was inherited, but after that Derived has relinquished all control over what happens. Once we're in process we're only able to see Base's member implementations. Polymorphism offers a way out, allowing us to redefine (in a derived class) members of a base class allowing these redefined members to be used from the base class's interface.

This is the essence of LSP: public inheritance should not be used to reuse the base class members (in derived classes) but to be reused (by the base class, polymorphically using derived class members reimplementing base class members).

Take a second to appreciate the implications of the above little program. The hello and process members aren't too impressive, but the implications of the example are. The process member could implement directory travel, hello could define the action to perform when encountering a file. Base::hello might simply show the name of a file, but Derived::hello might delete the file; might only list its name if its younger than a certain age; might list its name if it contains a certain text; etc., etc.. Up to now Derived would have to implement process's actions itself; Up to now code expecting a Base class reference or pointer could only perform Base's actions. Polymorphism allows us to reimplement members of base classes and to use those reimplemented members in code expecting base class references or pointers. Using polymorphism existing code may be reused by derived classes reimplementing the appropriate members of their base classes. It's about time to uncover how this magic can be realized.

Polymorphism, which is not the default in C++, solves the problem and allows the author of the classes to reach its goal. For the curious reader: prefix void hello() in the Base class with the keyword virtual and recompile. Running the modified program produces the intended and expected derived hello. Why this happens is explained next.

### 14.1 Virtual functions

By default the behavior of a member function called via a pointer or reference is determined by the implementation of that function in the pointer's or reference's class. E.g., a Vehicle * activates Vehicle's member functions, even when pointing to an object of a derived class. This is known as as early or static binding: the function to call is determined at compile-time. In C++ late or dynamic binding is realized using virtual member functions.

A member function becomes a virtual member function when its declaration starts with the keyword virtual. It is stressed once again that in C++, different from several other object oriented languages, this is not the default situation. By default static binding is used.

Once a function is declared virtual in a base class, it remains virtual in all derived classes. The keyword virtual should not be mentioned with members declared virtual in the base class. In derived classes those members should be provided with the override indicator.

In the vehicle classification system (see section 13.1), let's concentrate on the members mass and setMass. These members define the user interface of the class Vehicle. What we would like to accomplish is that this user interface can be used for Vehicle and for any class inheriting from Vehicle, since objects of those classes are themselves also Vehicles.

If we can define the user interface of our base class (e.g., Vehicle) such that it remains usable irrespective of the classes we derive from Vehicle our software achieves an enormous reusability: we design or software around Vehicle's user interface, and our software will also properly function for derived classes. Using plain inheritance doesn't accomplish this. If we define

```cpp
std::ostream &operator<<(std::ostream &out, Vehicle const &vehicle)
```
and Vehicle’s member mass returns 0, but Car’s member mass returns 1000, then twice a mass of 0 is reported when the following program is executed:

```cpp
int main()
{
    Vehicle vehicle;
    Car vw(1000);
    cout << vehicle << endl << vw << endl;
}
```

We’ve defined an overloaded insertion operator, but since it only knows about Vehicle’s user interface, ‘cout << vw’ will use vw’s Vehicle’s user interface as well, thus displaying a mass of 0.

Reusability is enhanced if we add a redefinable interface to the base class’s interface. A redefinable interface allows derived classes to fill in their own implementation, without affecting the user interface. At the same time the user interface will behave according to the derived class’s wishes, and not just to the base class’s default implementation.

Members of the reusable interface should be declared in the class’s private sections: conceptually they merely belong to their own classes (cf. section 14.7). In the base class these members should be declared virtual. These members can be redefined (overridden) by derived classes, and should there be provided with override indicators.

We keep our user interface (mass), and add the redefinable member vmass to Vehicle’s interface:

```cpp
class Vehicle
{
    public:
        int mass() const;
        int si_mass() const; // see below

    private:
        virtual vmass() const;
};
```

Separating the user interface from the redefinable interface is a sensible thing to do. It allows us to fine-tune the user interface (only one point of maintenance), while at the same time allowing us to standardize the expected behavior of the members of the redefinable interface. E.g., in many countries the International system of units is used, using the kilogram as the unit for mass. Some countries use other units (like the lbs: 1 kg being approx. 2.2046 lbs). By separating the user interface from the redefinable interface we can use one standard for the redefinable interface, and keep the flexibility of transforming the information ad-lib in the user interface.

Just to maintain a clean separation of user- and redefinable interface we might consider adding another accessor to Vehicle, providing the si_mass, simply implemented like this:

```cpp
int Vehicle::si_mass() const
{
    return out << "Vehicle’s mass is " << vehicle.mass() << " kg."
```
14.1. VIRTUAL FUNCTIONS

return vmass();
}

If \texttt{Vehicle} supports a member \texttt{d\_massFactor} then its \texttt{mass} member can be implemented like this:

```cpp
int Vehicle::mass()
{
    return d_massFactor * si_mass();
}
```

\texttt{Vehicle} itself could define \texttt{vmass} so that it returns a token value. E.g.,

```cpp
int Vehicle::vmass()
{
    return 0;
}
```

Now let’s have a look at the class \texttt{Car}. It is derived from \texttt{Vehicle}, and it inherits \texttt{Vehicle}’s user interface. It also has a data member \texttt{int d\_mass}, and it implements its own reusable interface:

```cpp
class Car: public Vehicle
{
...
    private:
        int vmass() override;
}
```

If \texttt{Car} constructors require us to specify the car’s mass (stored in \texttt{d\_mass}), then \texttt{Car} simply implements its \texttt{vmass} member like this:

```cpp
int Car::vmass() const
{
    return d_mass;
}
```

The class \texttt{Truck}, inheriting from \texttt{Car} needs two mass values: the tractor’s mass and the trailer’s mass. The tractor’s mass is passed to its \texttt{Car} base class, the trailer’s mass is passed to its \texttt{Vehicle d\_trailer} data member. \texttt{Truck}, too, overrides \texttt{vmass}, this time returning the sum of its tractor and trailer masses:

```cpp
int Truck::vmass() const
{
    return Car::si_mass() + d_trailer.si_mass();
}
```

Once a class member has been declared \texttt{virtual} it becomes a virtual member in all derived classes, whether or not these members are provided with the \texttt{override} indicator. But \texttt{override} \texttt{should} be used, as it allows to compiler to catch typos when writing down the derived class interface.

A member function may be declared \texttt{virtual anywhere} in a class hierarchy, but this probably defeats the underlying polymorphic class design, as the original base class is no longer capable of
completely covering the redefinable interfaces of derived classes. If, e.g., mass is declared virtual in Car, but not in Vehicle, then the specific characteristics of virtual member functions would only be available for Car objects and for objects of classes derived from Car. For a Vehicle pointer or reference static binding would remain to be used.

The effect of late binding (polymorphism) is illustrated below:

```cpp
void showInfo(Vehicle &vehicle)
{
    cout << "Info: " << vehicle << 'n';
}

int main()
{
    Car car(1200); // car with mass 1200
    Truck truck(6000, 115, // truck with cabin mass 6000,
                "Scania", 15000); // speed 115, make Scania,
                            // trailer mass 15000

    showInfo(car);       // see (1) below
    showInfo(truck);     // see (2) below

    Vehicle *vp = &truck;
    cout << vp->speed() << 'n'; // see (3) below
}
```

Now that mass is defined virtual, late binding is used:

- at (1), Car’s mass is displayed;
- at (2) Truck’s mass is displayed;
- at (3) a syntax error is generated. The member speed is not a member of Vehicle, and hence not callable via a Vehicle*.

The example illustrates that when a pointer to a class is used only the members of that class can be called. A member’s virtual characteristic only influences the type of binding (early vs. late), not the set of member functions that is visible to the pointer.

Through virtual members derived classes may redefine the behavior performed by functions called from base class members or from pointers or references to base class objects. This redefinition of base class members by derived classes is called overriding members.

### 14.2 Virtual destructors

When an object ceases to exist the object’s destructor is called. Now consider the following code fragment (cf. section 13.1):

```cpp
Vehicle *vp = new Land(1000, 120);
delete vp;             // object destroyed
```
14.3. PURE VIRTUAL FUNCTIONS

Here delete is applied to a base class pointer. As the base class defines the available interface delete vp calls ~Vehicle and ~Land remains out of sight. Assuming that Land allocates memory a memory leak results. Freeing memory is not the only action destructors can perform. In general they may perform any action that’s necessary when an object ceases to exist. But here none of the actions defined by ~Land are performed. Bad news...

In C++ this problem is solved by virtual destructors. A destructor can be declared virtual. When a base class destructor is declared virtual then the destructor of the actual class pointed to by a base class pointer bp is going to be called when delete bp is executed. Thus, late binding is realized for destructors even though the destructors of derived classes have unique names. Example:

```cpp
class Vehicle
{
    public:
        virtual ~Vehicle(); // all derived class destructors are
                                // now virtual as well.
};
```

By declaring a virtual destructor, the above delete operation (delete vp) correctly calls Land’s destructor, rather than Vehicle’s destructor.

Once a destructor is called it performs as usual, whether or not it is a virtual destructor. So, ~Land first executes its own statements and then calls ~Vehicle. Thus, the above delete vp statement uses late binding to call ~Vehicle and from this point on the object destruction proceeds as usual.

Destructors should always be defined virtual in classes designed as a base class from which other classes are going to be derived. Often those destructors themselves have no tasks to perform. In these cases the virtual destructor is given an empty body. For example, the definition of Vehicle::~Vehicle() may be as simple as:

```cpp
Vehicle::~Vehicle()
{}
```

Resist the temptation to define virtual destructors (even empty destructors) inline as this complicates class maintenance. Section 14.11 discusses the reason behind this rule of thumb.

14.3 Pure virtual functions

The base class Vehicle is provided with its own concrete implementations of its virtual members (mass and setMass). However, virtual member functions do not necessarily have to be implemented in base classes.

When the implementations of virtual members are omitted from base classes the class imposes requirements upon derived classes. The derived classes are required to provide the ‘missing implementations’.

This approach, in some languages (like C#, Delphi and Java) known as an interface, defines a protocol. Derived classes must obey the protocol by implementing the as yet not implemented members. If a class contains at least one member whose implementation is missing no objects of that class can be defined.

Such incompletely defined classes are always base classes. They enforce a protocol by merely declaring names, return values and arguments of some of their members. These classes are call abstract
classes or abstract base classes. Derived classes become non-abstract classes by implementing the as yet not implemented members.

Abstract base classes are the foundation of many design patterns (cf. Gamma et al. (1995)), allowing the programmer to create highly reusable software. Some of these design patterns are covered by the C++ Annotations (e.g., the Template Method in section 24.2), but for a thorough discussion of design patterns the reader is referred to Gamma et al.’s book.

Members that are merely declared in base classes are called pure virtual functions. A virtual member becomes a pure virtual member by postfixing \( = 0 \) to its declaration (i.e., by replacing the semi-colon ending its declaration by ‘\( = 0; \)’). Example:

```
#include <iosfwd>
class Base
{
  public:
    virtual ~Base();
    virtual std::ostream &insertInto(std::ostream &out) const = 0;
};
inline std::ostream &operator<<(std::ostream &out, Base const &base)
{
  return base.insertInto(out);
}
```

All classes derived from `Base` **must** implement the `insertInto` member function, or their objects cannot be constructed. This is neat: all objects of class types derived from `Base` can now always be inserted into `ostream` objects.

Could the virtual destructor of a base class ever be a pure virtual function? The answer to this question is no. First of all, there is no need to enforce the availability of destructors in derived classes as destructors are provided by default (unless a destructor is declared with the \= delete attribute using the new C++11 standard). Second, if it is a pure virtual member its implementation does not exist. However, derived class destructors eventually call their base class destructors. How could they call base class destructors if their implementations are lacking? More about this in the next section.

Often, but not necessarily, pure virtual member functions are **const** member functions. This allows the construction of constant derived class objects. In other situations this might not be necessary (or realistic), and non-constant member functions might be required. The general rule for **const** member functions also applies to pure virtual functions: if the member function alters the object’s data members, it cannot be a **const** member function.

Abstract base classes frequently don’t have data members. However, once a base class declares a pure virtual member it **must** be declared identically in derived classes. If the implementation of a pure virtual function in a derived class alters the derived class object’s data, then that function cannot be declared as a **const** member. Therefore, the author of an abstract base class should carefully consider whether a pure virtual member function should be a **const** member function or not.

### 14.3.1 Implementing pure virtual functions

Pure virtual member functions may be implemented. To implement a pure virtual member function, provide it with its normal \( = 0; \) specification, but implement it as well. Since the \( = 0; \) ends in
14.3. PURE VIRTUAL FUNCTIONS

a semicolon, the pure virtual member is always at most a declaration in its class, but an implement-
mentation may either be provided outside from its interface (maybe using \texttt{inline}).

Pure virtual member functions may be called from derived class objects or from its class or derived
class members by specifying the base class and scope resolution operator together with the member
to be called. Example:

```cpp
#include <iostream>

class Base
{
    public:
        virtual ~Base();
        virtual void pureimp() = 0;
};
Base::~Base()
{}
void Base::pureimp()
{
    std::cout << "Base::pureimp() called\n";
}
class Derived: public Base
{
    public:
        virtual void pureimp();
};
inline void Derived::pureimp()
{
    Base::pureimp();
    std::cout << "Derived::pureimp() called\n";
}
int main()
{
    Derived derived;
    derived.pureimp();
    derived.Base::pureimp();

    Derived *dp = &derived;
    dp->pureimp();
    dp->Base::pureimp();
}
```

Implementing a pure virtual member has limited use. One could argue that the pure virtual member
function's implementation may be used to perform tasks that can already be performed at the base
class level. However, there is no guarantee that the base class virtual member function is actually
CHAPTER 14. POLYMORPHISM

going to be called. Therefore base class specific tasks could as well be offered by a separate member, without blurring the distinction between a member doing some work and a pure virtual member enforcing a protocol.

14.4 Explicit virtual overrides

Consider the following situations:

• A class Value is a value class. It offers a copy constructor, an overloaded assignment operator, maybe move operations, and a public, non-virtual constructor. In section 14.7 it is argued that such classes is not suited as a base class. New classes should not inherit from Value. How to enforce this?

• A polymorphic class Base defines a virtual member v_process(int32_t). A class derived from Base needs to override this member, but the author mistakenly defined v_process(int32_t). How to prevent such errors, breaking the polymorphic behavior of the derived class?

• A class Derived, derived from a polymorphic Base class overrides the member Base::v_process, but classes that are in turn derived from Derived should no longer override v_process, but may override other virtual members like v_call and v_display. How to enforce this restricted polymorphic character for classes derived from Derived?

C++11 allows the use of two special identifiers, final and override to realize the above. These identifiers are special in the sense that they only require their special meanings in specific contexts. Outside of this context they are just plain identifiers, allowing the programmer to define a variable like bool final.

The identifier final can be applied to class declarations to indicate that the class cannot be used as a base class. E.g.:

```cpp
class Base1 final // cannot be a base class
{);
class Derived1: public Base1 // ERR: Base1 is final
{);

class Base2 // OK as base class
{);
class Derived2 final: public Base2 // OK, but Derived2 can't be
{); // used as a base class

class Derived: public Derived2 // ERR: Derived2 is final
{);
```

The identifier final can also be added to virtual member declarations. This indicates that those virtual members cannot be overridden by derived classes. The restricted polymorphic character of a class, mentioned above, can thus be realized as follows:

```cpp
class Base
{
    virtual int v_process(); // define polymorphic behavior
    virtual int v_call();
    virtual int v_display();
```
14.5 Virtual functions and multiple inheritance

In chapter 6 we encountered the class `fstream`, one class offering features of `ifstream` and `ofstream`. In chapter 13 we learned that a class may be derived from multiple base classes. Such a derived class inherits the properties of all its base classes. Polymorphism can also be used in combination with multiple inheritance.

Consider what would happen if more than one ‘path’ leads from the derived class up to its (base) classes. This is illustrated in the next (fictitious) example where a class `Derived` is doubly derived from `Base`:

```cpp
class Base
{
    int d_field;
public:
    void setfield(int val);
    int field() const;
};
inline void Base::setfield(int val)
{
    d_field = val;
}
inline int Base::field() const
```
CHAPTER 14. POLYMORPHISM

Figure 14.1: Duplication of a base class in multiple derivation.

```cpp

{ return d_field;
}

class Derived: public Base, public Base
{
};
```

Due to the double derivation, Base's functionality now occurs twice in Derived. This results in ambiguity: when the function `setfield()` is called for a Derived class object, which function will that be as there are two of them? The scope resolution operator won't come to the rescue and so the C++ compiler cannot compile the above example and (correctly) identifies an error.

The above code clearly duplicates its base class in the derivation, which can of course easily be avoided by not doubly deriving from Base (or by using composition (!)). But duplication of a base class can also occur through nested inheritance, where an object is derived from, e.g., a Car and from an Air (cf. section 13.1). Such a class would be needed to represent, e.g., a flying car\(^2\). An AirCar would ultimately contain two Vehicles, and hence two mass fields, two `setMass()` functions and two `mass()` functions. Is this what we want?

### 14.5.1 Ambiguity in multiple inheritance

Let's investigate closer why an AirCar introduces ambiguity, when derived from Car and Air.

- An AirCar is a Car, hence a Land, and hence a Vehicle.
- However, an AirCar is also an Air, and hence a Vehicle.

The duplication of Vehicle data is further illustrated in Figure 14.1. The internal organization of an AirCar is shown in Figure 14.2. The C++ compiler detects the ambiguity in an AirCar object, and will therefore not compile statements like:

```cpp

AirCar jBond;
cout << jBond.mass() << '\n';
```

Which member function `mass` to call cannot be determined by the compiler but the programmer has two possibilities to resolve the ambiguity for the compiler:

\(^2\)such as the one in James Bond vs. the Man with the Golden Gun...
14.5. VIRTUAL FUNCTIONS AND MULTIPLE INHERITANCE

Figure 14.2: Internal organization of an AirCar object.

- First, the function call where the ambiguity originates can be modified. The ambiguity is resolved using the scope resolution operator:

```cpp
// let’s hope that the mass is kept in the Car
// part of the object..
cout << jBond.Car::mass() << ‘\n’;
```

The scope resolution operator and the class name are put right before the name of the member function.

- Second, a dedicated function `mass` could be created for the class `AirCar`:

```cpp
int AirCar::mass() const
{
    return Car::mass();
}
```

The second possibility is preferred as it does not require the compiler to flag an error; nor does it require the programmer using the class `AirCar` to take special precautions.

However, there exists a more elegant solution, discussed in the next section.

14.5.2 Virtual base classes

As illustrated in Figure 14.2, an AirCar represents two Vehicles. This not only results in an ambiguity about which function to use to access the mass data, but it also defines two mass fields in an AirCar. This is slightly redundant, since we can assume that an AirCar has but one mass.

It is, however, possible to define an AirCar as a class consisting of but one Vehicle and yet using multiple derivation. This is realized by defining the base classes that are multiply mentioned in a derived class’s inheritance tree as a virtual base class.

For the class `AirCar` this implies a small change when deriving an AirCar from `Land` and `Air` classes:

```cpp
class Land: virtual public Vehicle
{
    // etc
};

class Car: public Land
```
Virtual derivation ensures that a `Vehicle` is only added once to a derived class. This means that the route along which a `Vehicle` is added to an `AirCar` is no longer depending on its direct base classes; we can only state that an `AirCar` is a `Vehicle`. The internal organization of an `AirCar` after virtual derivation is shown in Figure 14.3.

When a class `Third` inherits from a base class `Second` which in turn inherits from a base class `First` then the `First` class constructor called by the `Second` class constructor is also used when this `Second` constructor is used when constructing a `Third` object. Example:

```cpp
class First
{
  public:
    First(int x);
};
class Second: public First
{
  public:
    Second(int x)
    :
      First(x)
    {}
};
class Third: public Second
{
  public:
    Third(int x)
    :
      Second(x) // calls First(x)
    {}
};
```
14.5. VIRTUAL FUNCTIONS AND MULTIPLE INHERITANCE

The above no longer holds true when `Second` uses virtual derivation. When `Second` uses virtual derivation its base class constructor is ignored when `Second`'s constructor is called from `Third`. Instead `Second` by default calls `First`'s default constructor. This is illustrated by the next example:

```cpp
class First
{
public:
    First()
    {
        cout << "First()\n";
    }
    First(int x);
};
class Second: public virtual First  // note: virtual
{
public:
    Second(int x)
    :
        First(x)
    {}
};
class Third: public Second
{
public:
    Third(int x)
    :
        Second(x)
    {}
};
int main()
{
    Third third(3);  // displays 'First()'
}
```

When constructing `Third` `First`'s default constructor is used by default. `Third`'s constructor, however, may overrule this default behavior by explicitly specifying the constructor to use. Since the `First` object must be available before `Second` can be constructed it must be specified first. To call `First(int)` when constructing `Third(int)` the latter constructor can be defined as follows:

```cpp
class Third: public Second
{
public:
    Third(int x)
    :
        First(x), // now First(int) is called.
        Second(x)
    {}
};
```

This behavior may seem puzzling when simple linear inheritance is used but it makes sense when multiple inheritance is used with base classes using virtual inheritance. Consider `AirCar`: when
Air and Car both virtually inherit from Vehicle will Air and Car both initialize the common Vehicle object? If so, which one is going to be called first? What if Air and Car use different Vehicle constructors? All these questions can be avoided by passing the responsibility for the initialization of a common base class to the class eventually using the common base class object. In the above example Third. Hence Third is provided an opportunity to specify the constructor to use when initializing First.

Multiple inheritance may also be used to inherit from classes that do not all use virtual inheritance. Assume we have two classes, Derived1 and Derived2, both (possibly virtually) derived from Base.

We now address the question which constructors will be called when calling a constructor of the class Final: public Derived1, public Derived2.

To distinguish the involved constructors Base1 indicates the Base class constructor called as base class initializer for Derived1 (and analogously: Base2 called from Derived2). A plain Base indicates Base’s default constructor.

Derived1 and Derived2 indicate the base class initializers used when constructing a Final object.

Now we’re ready to distinguish the various cases when constructing an object of the class Final: public Derived1, public Derived2:

- classes:

  Derived1: public Base
  Derived2: public Base

  This is normal, non virtual multiple derivation. The following constructors are called in the order shown:

  Base1,
  Derived1,
  Base2,
  Derived2

- classes:

  Derived1: public Base
  Derived2: virtual public Base

  Only Derived2 uses virtual derivation. Derived2’s base class constructor is ignored. Instead, Base is called and it is called prior to any other constructor:

  Base,
  Base1,
  Derived1,
  Derived2

  As only one class uses virtual derivation, two Base class objects remain available in the eventual Final class.

- classes:

  Derived1: virtual public Base
  Derived2: public Base
Only Derived1 uses virtual derivation. Derived1’s base class constructor is ignored. Instead, Base is called and it is called prior to any other constructor. Different from the first (non-virtual) case Base is now called, rather than Base1:

Base,
Derived1,
Base2,
Derived2

- classes:

Derived1: virtual public Base
Derived2: virtual public Base

Both base classes use virtual derivation and so only one Base class object will be present in the Final class object. The following constructors are called in the order shown:

Base,
Derived1,
Derived2

Virtual derivation is, in contrast to virtual functions, a pure compile-time issue. Virtual inheritance merely defines how the compiler defines a class’s data organization and construction process.

### 14.5.3 When virtual derivation is not appropriate

Virtual inheritance can be used to merge multiply occurring base classes. However, situations may be encountered where multiple occurrences of base classes is appropriate. Consider the definition of a Truck (cf. section 13.5):

```cpp
class Truck: public Car
{
    int d_trailer_mass;

public:
    Truck();
    Truck(int engine_mass, int sp, char const *nm,
        int trailer_mass);
    void setMass(int engine_mass, int trailer_mass);
    int mass() const;
};
Truck::Truck(int engine_mass, int sp, char const *nm,
    int trailer_mass)
:
    Car(engine_mass, sp, nm)
{
    d_trailer_mass = trailer_mass;
}
int Truck::mass() const
{
    return // sum of:
        Car::mass() + // engine part plus
```
This definition shows how a Truck object is constructed to contain two mass fields: one via its derivation from Car and one via its own int d_trailer_mass data member. Such a definition is of course valid, but it could also be rewritten. We could derive a Truck from a Car and from a Vehicle, thereby explicitly requesting the double presence of a Vehicle; one for the mass of the engine and cabin, and one for the mass of the trailer. A slight complication is that a class organization like

```cpp
class Truck: public Car, public Vehicle
```

is not accepted by the C++ compiler. As a Vehicle is already part of a Car, it is therefore not needed once again. This organization may, however be forced using a small trick. By creating an additional class inheriting from Vehicle and deriving Truck from that additional class rather than directly from Vehicle the problem is solved. Simply derive a class TrailerVeh from Vehicle, and then Truck from Car and TrailerVeh:

```cpp
class TrailerVeh: public Vehicle
{
    public:
    TrailerVeh(int mass) :
        Vehicle(mass)
    {}
};
class Truck: public Car, public TrailerVeh
{
    public:
    Truck();
    Truck(int engine_mass, int sp, char const *nm, int trailer_mass);
    void setMass(int engine_mass, int trailer_mass);
    int mass() const;
};
inline Truck::Truck(int engine_mass, int sp, char const *nm, int trailer_mass)
:
    Car(engine_mass, sp, nm),
    TrailerVeh(trailer_mass)
{}
inline int Truck::mass() const
{
    return // sum of:
        Car::mass() + // engine part plus
        TrailerVeh::mass(); // the trailer
}
```

14.6 Run-time type identification

C++ offers two ways to (run-time) retrieve the type of objects and expressions. The possibilities of C++'s run-time type identification are limited compared to languages like Java. Usually static
type checking and static type identification is used in C++. Static type checking is possibly safer and certainly more efficient than run-time type identification and should therefore be preferred over run-time type identification. But situations exist where run-time type identification is appropriate. C++ offers run-time type identification through the *dynamic cast* and *typeid* operators.

- A dynamic cast is used to convert a base class pointer or reference to a derived class pointer or reference. This is also known as *down-casting*.
- The typeid operator returns the actual type of an expression.

These operators can be used with objects of classes having at least one virtual member function.

### 14.6.1 The dynamic_cast operator

The `dynamic_cast<>` operator is used to convert a base class pointer or reference to, respectively, a derived class pointer or reference. This is also called *down-casting* as direction of the cast is *down* the inheritance tree.

A dynamic cast’s actions are determined run-time; it can only be used if the base class declares at least one virtual member function. For the dynamic cast to succeed, the destination class’s Vtable must be equal to the Vtable to which the dynamic cast’s argument refers to, lest the cast fails and returns 0 (if a dynamic cast of a pointer was requested) or throws a `std::bad_cast` exception (if a dynamic cast of a reference was requested).

In the following example a pointer to the class `Derived` is obtained from the `Base` class pointer `bp`:

```cpp
class Base
{
  public:
    virtual ~Base();
};
class Derived: public Base
{
  public:
    char const *toString();
};
inline char const *Derived::toString()
{
  return "Derived object";
}
int main()
{
  Base *bp;
  Derived *dp,
  Derived d;
  bp = &d;
  dp = dynamic_cast<Derived*>(bp);
  if (dp)
    cout << dp->toString() << '\n';
  else
```
In the condition of the above if statement the success of the dynamic cast is verified. This verification is performed at run-time, as the actual class of the objects to which the pointer points is only known by then.

If a base class pointer is provided, the dynamic cast operator returns 0 on failure and a pointer to the requested derived class on success.

Assume a vector<Base *> is used. Such a vector's pointers may point to objects of various classes, all derived from Base. A dynamic cast returns a pointer to the specified class if the base class pointer indeed points to an object of the specified class and returns 0 otherwise.

We could determine the actual class of an object a pointer points to by performing a series of checks to find the derived class to which a base class pointer points. Example:

```cpp
class Base
{
    public:
    virtual ~Base();
};
class Derived1: public Base;
class Derived2: public Base;

int main()
{
    vector<Base *> vb(initializeBase());
    Base *bp = vb.front();
    if (dynamic_cast<Derived1 *>(bp))
        cout << "bp points to a Derived1 class object\n";
    else if (dynamic_cast<Derived2 *>(bp))
        cout << "bp points to a Derived2 class object\n";
}
```

Alternatively, a reference to a base class object may be available. In this case the dynamic_cast operator throws an exception if the down casting fails. Example:

```cpp
#include <iostream>
#include <typeinfo>

class Base
{
    public:
    virtual ~Base();
    virtual char const*toString();
};
inline char const*Base::toString()
{
    return "Base::toString() called";
}
class Derived1: public Base
```
class Derived2: public Base
{
};

Base::~Base()
{
}

void process(Base &b)
{
  try
  {
    std::cout << dynamic_cast<Derived1 &>(b).toString() << '\n';
  } catch (std::bad_cast) {}
  try
  {
    std::cout << dynamic_cast<Derived2 &>(b).toString() << '\n';
  } catch (std::bad_cast) {
    std::cout << "Bad cast to Derived2\n";
  }
}

int main()
{
  Derived1 d;
  process(d);
}

/*
Generated output:
Base::toString() called
Bad cast to Derived2
*/

In this example the value std::bad_cast is used. A std::bad_cast exception is thrown if the
dynamic cast of a reference to a derived class object fails.

Note the form of the catch clause: bad_cast is the name of a type. Section 17.4.1 describes how
such a type can be defined.

The dynamic cast operator is a useful tool when an existing base class cannot or should not be
modified (e.g., when the sources are not available), and a derived class may be modified instead.
Code receiving a base class pointer or reference may then perform a dynamic cast to the derived
class to access the derived class's functionality.

You may wonder in what way the behavior of the dynamic_cast differs from that of the
static_cast.

When the static_cast is used, we tell the compiler that it must convert a pointer or reference
to its expression type to a pointer or reference of its destination type. This holds true whether the
base class declares virtual members or not. Consequently, all the static_cast’s actions can be
determined by the compiler, and the following compiles fine:

class Base
{  // maybe or not virtual members
};
class Derived1: public Base
{};
class Derived2: public Base
{};

int main()
{
    Derived1 derived1;
    Base *bp = &derived1;
    Derived1 &d1ref = static_cast<Derived1 &>(*bp);
    Derived2 &d2ref = static_cast<Derived2 &>(*bp);
}

Pay attention to the second static_cast: here the Base class object is cast to a Derived2 class reference. The compiler has no problems with this, as Base and Derived2 are related by inheritance.

Semantically, however, it makes no sense as bp in fact points to a Derived1 class object. This is detected by a dynamic_cast. A dynamic_cast, like the static_cast, converts related pointer or reference types, but the dynamic_cast provides a run-time safeguard. The dynamic cast fails when the requested type doesn’t match the actual type of the object we’re pointing at. In addition, the dynamic_cast’s use is much more restricted than the static_cast’s use, as the dynamic_cast can only be used for downcasting to derived classes having virtual members.

In the end a dynamic cast is a cast, and casts should be avoided whenever possible. When the need for dynamic casting arises ask yourself whether the base class has correctly been designed. In situations where code expects a base class reference or pointer the base class interface should be all that is required and using a dynamic cast should not be necessary. Maybe the base class’s virtual interface can be modified so as to prevent the use of dynamic casts. Start frowning when encountering code using dynamic casts. When using dynamic casts in your own code always properly document why the dynamic cast was appropriately used and was not avoided (for an example where a dynamic cast is used on purpose, see section 24.8.3).

14.6.2 The ‘typeid’ operator

As with the dynamic_cast operator, typeid is usually applied to references to base class objects that refer to derived class objects. Typeid should only be used with base classes offering virtual members.

Before using typeid the <typeinfo> header file must be included.

The typeid operator returns an object of type type_info. Different compilers may offer different implementations of the class type_info, but at the very least typeid must offer the following interface:

    class type_info
    {
        public:
            virtual ~type_info();
            int operator==(type_info const &other) const;
14.6. RUN-TIME TYPE IDENTIFICATION

```cpp
type_info operator!=(type_info const &other) const;
bool before(type_info const &rhs) const;
char const *name() const;
private:
type_info(type_info const &other);
type_info &operator=(type_info const &other);
```

Note that this class has a private copy constructor and a private overloaded assignment operator. This prevents code from constructing `type_info` objects and prevents code from assigning `type_info` objects to each other. Instead, `type_info` objects are constructed and returned by the `typeid` operator.

If the `typeid` operator is passed a base class reference it is able to return the actual name of the type the reference refers to. Example:

```cpp
class Base;
class Derived: public Base;

Derived d;
Base &br = d;

cout << typeid(br).name() << '
';
```

In this example the `typeid` operator is given a base class reference. It prints the text “Derived”, being the class name of the class `br` actually refers to. If `Base` does not contain virtual functions, the text “Base” is printed.

The `typeid` operator can be used to determine the name of the actual type of expressions, not just of class type objects. For example:

```cpp
cout << typeid(12).name() << '
';  // prints: int
cout << typeid(12.23).name() << '
';  // prints: double
```

Note, however, that the above example is suggestive at most. It may print `int` and `double`, but this is not necessarily the case. If portability is required, make sure no tests against these static, built-in text-strings are required. Check out what your compiler produces in case of doubt.

In situations where the `typeid` operator is applied to determine the type of a derived class, a base class reference should be used as the argument of the `typeid` operator. Consider the following example:

```cpp
class Base;  // contains at least one virtual function
class Derived: public Base;

Base *bp = new Derived;  // base class pointer to derived object
if (typeid(bp) == typeid(Derived *))  // 1: false
    ...
if (typeid(bp) == typeid(Base *))  // 2: true
    ...
if (typeid(bp) == typeid(Derived))  // 3: false
    ...
if (typeid(bp) == typeid(Base))  // 4: false
```
if (typeid(*bp) == typeid(Derived)) // 5: true
...
if (typeid(*bp) == typeid(Base)) // 6: false
...
Base &br = *bp;
if (typeid(br) == typeid(Derived)) // 7: true
...
if (typeid(br) == typeid(Base)) // 8: false
...

Here, (1) returns false as a Base * is not a Derived *. (2) returns true, as the two pointer types are the same, (3) and (4) return false as pointers to objects are not the objects themselves.

On the other hand, if *bp is used in the above expressions, then (1) and (2) return false as an object (or reference to an object) is not a pointer to an object, whereas (5) now returns true: *bp actually refers to a Derived class object, and typeid(*bp) returns typeid(Derived). A similar result is obtained if a base class reference is used: 7 returning true and 8 returning false.

The type_info::before(type_info const &rhs) member is used to determine the collating order of classes. This is useful when comparing two types for equality. The function returns a nonzero value if *this precedes rhs in the hierarchy or collating order of the used types. When a derived class is compared to its base class the comparison returns 0, otherwise a non-zero value. E.g.:

cout << typeid(ifstream).before(typeid(istream)) << '
' << // not 0
typeid(istream).before(typeid(ifstream)) << '
'; // 0

With built-in types the implementor may implement that non-0 is returned when a ‘wider’ type is compared to a ‘smaller’ type and 0 otherwise:

cout << typeid(double).before(typeid(int)) << '
' << // not 0
typeid(int).before(typeid(double)) << '
'; // 0

When two equal types are compared, 0 is returned:

cout << typeid(ifstream).before(typeid(ifstream)) << '
'; // 0

When a 0-pointer is passed to the operator typeid a bad_typeid exception is thrown.

14.7 Inheritance: when to use to achieve what?

Inheritance should not be applied automatically and thoughtlessly. Often composition can be used instead, improving on a class’s design by reducing coupling. When inheritance is used public inheritance should not automatically be used but the type of inheritance that is selected should match the programmer’s intent.

We’ve seen that polymorphic classes on the one hand offer interface members defining the functionality that can be requested of base classes and on the other hand offer virtual members that can be overridden. One of the signs of good class design is that member functions are designed according
to the principle of ‘one function, one task’. In the current context: a class member should either be a member of the class’s public or protected interface or it should be available as a virtual member for reimplementation by derived classes. Often this boils down to virtual members that are defined in the base class’s private section. Those functions shouldn’t be called by code using the base class, but they exist to be overridden by derived classes using polymorphism to redefine the base class’s behavior.

The underlying principle was mentioned before in the introductory paragraph of this chapter: according to the Liskov Substitution Principle (LSP) an is-a relationship between classes (indicating that a derived class object is a base class object) implies that a derived class object may be used in code expecting a base class object.

In this case inheritance is used not to let the derived class use the facilities already implemented by the base class but to reuse the base class polymorphically by reimplementing the base class’s virtual members in the derived class.

In this section we’ll discuss the reasons for using inheritance. Why should inheritance (not) be used? If it is used what do we try to accomplish by it?

Inheritance often competes with composition. Consider the following two alternative class designs:

```cpp
class Derived: // derived from Base
{
    ...
};

class Composed
{
    Base d_base;
    ...
};
```

Why and when prefer Derived over Composed and vice versa? What kind of inheritance should be used when designing the class Derived?

- Since Composed and Derived are offered as alternatives we are looking at the design of a class (Derived or Composed) that is implemented-in-terms-of another class.

- Since Composed does itself not make Base’s interface available, Derived shouldn’t do so either. The underlying principle is that private inheritance should be used when deriving a class Derived from Base where Derived is implemented-in-terms-of Base.

- Should we use inheritance or composition? Here are some arguments:
  - In general terms composition results in looser coupling and should therefore be preferred over inheritance.
  - Composition allows us to define classes having multiple members of the same type (think about a class having multiple std::string members) which can not be realized using inheritance.
  - Composition allows us to separate the class’s interface from its implementation. This allows us to modify the class’s data organization without the need to recompile code using our class. This is also known as the bridge design pattern or the compiler firewall or pimpl (pointer to the implementation) idiom.
  - If Base offers members in its protected interface that must be used when implementing Derived inheritance must also be used. Again: since we’re implementing-in-terms-of the inheritance type should be private.
Protected inheritance may be considered when the derived class (D) itself is intended as a base class that should only make the members of its own base class (B) available to classes that are derived from it (i.e., D).

Private inheritance should also be used when a derived class is-a certain type of base class, but in order to initialize that base class an object of another class type must be available. Example: a new istream class-type (say: a stream IRandStream from which random numbers can be extracted) is derived from std::istream. Although an istream can be constructed empty (receiving its streambuf later using its rdbuf member), it is clearly preferable to initialize the istream base class right away.

Assuming that a Randbuffer: public std::streambuf has been created for generating random numbers then IRandStream can be derived from Randbuffer and std::istream. That way the istream base class can be initialized using the Randbuffer base class.

As a RandStream is definitely not a Randbuffer public inheritance is not appropriate. In this case IRandStream is-implemented-in-terms-of a Randbuffer and so private inheritance should be used. IRandStream’s class interface should therefore start like this:

```cpp
class IRandStream: private Randbuffer, public std::istream
{
    public:
        IRandStream(int lowest, int highest) // defines the range
            :
            Randbuffer(lowest, highest),
            std::istream(this) // passes &Randbuffer
            {} // defines the range
    ... }
};
```

Public inheritance should be reserved for classes for which the LSP holds true. In those cases the derived classes can always be used instead of the base class from which they derive by code merely using base class references, pointers or members (i.e., conceptually the derived class is-a base class). This most often applies to classes derived from base classes offering virtual members. To separate the user interface from the redefinable interface the base class’s public interface should not contain virtual members (except for the virtual destructor) and the virtual members should all be in the base class’s private section. Such virtual members can still be overridden by derived classes (this should not come as a surprise, considering how polymorphism is implemented) and this design offers the base class full control over the context in which the redefined members are used. Often the public interface merely calls a virtual member, but those members can always be redefined to perform additional duties.

The prototypical form of a base class therefore looks like this:

```cpp
class Base
{
    public:
        virtual ~Base(); // calls virtual members (e.g., // v_process)
        void process(); // calls virtual members (e.g., // v_process)
    private:
        virtual void v_process(); // overridden by derived classes
};
```
Alternatively a base class may offer a non-virtual destructor, which should then be protected. It shouldn’t be public to prevent deleting objects through their base class pointers (in which case virtual destructors should be used). It should be protected to allow derived class destructors to call their base class destructors. Such base classes should, for the same reasons, have non-public constructors and overloaded assignment operators.

14.8 The ‘streambuf’ class

The class `std::streambuf` receives the character sequences processed by streams and defines the interface between stream objects and devices (like a file on disk). A `streambuf` object is usually not directly constructed, but usually it is used as base class of some derived class implementing the communication with some concrete device.

The primary reason for existence of the class `streambuf` is to decouple the `stream` classes from the devices they operate upon. The rationale here is to add an extra layer between the classes allowing us to communicate with devices and the devices themselves. This implements a *chain of command* which is seen regularly in software design.

The *chain of command* is considered a generic pattern when designing reusable software, encountered also in, e.g., the TCP/IP stack.

A `streambuf` can be considered yet another example of the chain of command pattern. Here the program talks to `stream` objects, which in turn forward their requests to `streambuf` objects, which in turn communicate with the devices. Thus, as we will see shortly, we are able to do in user-software what had to be done via (expensive) system calls before.

The class `streambuf` has no public constructor, but does make available several public member functions. In addition to these public member functions, several member functions are only available to classes derived from `streambuf`. In section 14.8.2 a predefined specialization of the class `streambuf` is introduced. All public members of `streambuf` discussed here are also available in `filebuf`.

The next section shows the `streambuf` members that may be overridden when deriving classes from `streambuf`. Chapter 24 offers concrete examples of classes derived from `streambuf`.

The class `streambuf` is used by streams performing input operations and by streams performing output operations and their member functions can be ordered likewise. The type `std::streamsize` used below may, for all practical purposes, be considered equal to the type `size_t`.

When inserting information into `ostream` objects the information is eventually passed on to the `ostream’s` `streambuf`. The `streambuf` may decide to throw an exception. However, this exception does not leave the `ostream` *using* the `streambuf`. Rather, the exception is caught by the `ostream`, which sets its `ios::bad_bit`. Exception raised by manipulators inserted into `ostream` objects are not caught by the `ostream` objects.

**Public members for input operations**

- `std::streamsize in_avail()`:

  Returns a lower bound on the number of characters that can be read immediately.

- `int sbumpc()`:

  The next available character or `EOF` is returned. The returned character is removed from the `streambuf` object. If no input is available, `sbumpc` calls the (protected)
CHAPTER 14. POLYMORPHISM

member uflow (see section 14.8.1 below) to make new characters available. EOF is returned if no more characters are available.

- int sgetc():

  The next available character or EOF is returned. The character is not removed from the streambuf object. To remove a character from the streambuf object, sbumpc (or sgetn) can be used.

- int sgetn(char *buffer, std::streamsize n):

  At most n characters are retrieved from the input buffer, and stored in buffer. The actual number of characters read is returned. The (protected) member xsgetn (see section 14.8.1 below) is called to obtain the requested number of characters.

- int snextc():

  The current character is obtained from the input buffer and returned as the next available character or EOF is returned. The character is not removed from the streambuf object.

- int sputback(char c):

  Inserts c into the streambuf's buffer to be returned as the next character to read from the streambuf object. Caution should be exercised when using this function: often there is a maximum of just one character that can be put back.

- int sungetc():

  Returns the last character read to the input buffer, to be read again at the next input operation. Caution should be exercised when using this function: often there is a maximum of just one character that can be put back.

Public members for output operations

- int pubsync():

  Synchronizes (i.e., flush) the buffer by writing any information currently available in the streambuf's buffer to the device. Normally only used by classes derived from streambuf.

- int sputc(char c):

  Character c is inserted into the streambuf object. If, after writing the character, the buffer is full, the function calls the (protected) member function overflow to flush the buffer to the device (see section 14.8.1 below).

- int sputn(char const *buffer, std::streamsize n):

  At most n characters from buffer are inserted into the streambuf object. The actual number of characters inserted is returned. This member function calls the (protected) member xsputn (see section 14.8.1 below) to insert the requested number of characters.

Public members for miscellaneous operations

The next three members are normally only used by classes derived from streambuf.
14.8. THE ‘STREAMBUF’ CLASS

- `ios::pos_type pubseekoff(ios::off_type offset, ios::seekdir way, ios::openmode mode = ios::in | ios::out)`:
  Sets the offset of the next character to be read or written to `offset`, relative to the standard `ios::seekdir` values indicating the direction of the seeking operation.

- `ios::pos_type pubseekpos(ios::pos_type offset, ios::openmode mode = ios::in | ios::out)`:
  Sets the absolute position of the next character to be read or written to `pos`.

- `streambuf *pubsetbuf(char* buffer, std::streamsize n)`:
  The `streambuf` object is going to use the buffer accommodating at least `n` characters.

14.8.1 Protected ‘streambuf’ members

The protected members of the class `streambuf` are important for understanding and using `streambuf` objects. Although there are both protected data members and protected member functions defined in the class `streambuf` the protected data members are not mentioned here as using them would violate the principle of data hiding. As `streambuf`'s set of member functions is quite extensive, it is hardly ever necessary to use its data members directly. The following subsections do not even list all protected member functions but only those are covered that are useful for constructing specializations.

`Streambuf` objects control a buffer, used for input and/or output, for which begin-, actual- and end-pointers have been defined, as depicted in figure 14.4.

`Streambuf` offers one protected constructor:

- `streambuf::streambuf()`:
  Default (protected) constructor of the class `streambuf`.

14.8.1.1 Protected members for input operations

Several protected member functions are available for input operations. The member functions marked `virtual` may or course be redefined in derived classes:

- `char *eback()`:
  `Streambuf` maintains three pointers controlling its input buffer: `eback` points to the ‘end of the putback’ area: characters can safely be put back up to this position. See also figure 14.4. `Eback` points to the `beginning` of the input buffer.

- `char *egptr()`:
  `Egptr` points just beyond the last character that can be retrieved from the input buffer. See also figure 14.4. If `gptr` equals `egptr` the buffer must be refilled. This should be implemented by calling `underflow`, see below.

- `void gbump(int n)`:
  The object’s `gptr` (see below) is advanced over `n` positions.
Figure 14.4: Input- and output buffer pointers of the class 'streambuf'
• char *gptr():

    Gptr points to the next character to be retrieved from the object’s input buffer. See also figure 14.4.

• virtual int pbackfail(int c):

    This member function may be overridden by derived classes to do something intelligent when putting back character c fails. One might consider restoring the old read pointer when input buffer’s begin has been reached. This member function is called when ungetting or putting back a character fails. In particular, it is called when

    - gptr() == 0: no buffering used,
    - gptr() == eback(): no more room to push back,
    - *gptr() != c: a different character than the next character to be read must be pushed back.

    If c == endOfFile() then the input device must be reset by one character position. Otherwise c must be prepended to the characters to be read. The function should return EOF on failure. Otherwise 0 can be returned.

• void setg(char *beg, char *next, char *beyond):

    Initializes an input buffer. beg points to the beginning of the input area, next points to the next character to be retrieved, and beyond points to the location just beyond the input buffer’s last character. Usually next is at least beg + 1, to allow for a putback operation. No input buffering is used when this member is called as setg(0, 0, 0). See also the member uflow, below.

• virtual streamsize showmanyc():

    (Pronounce: s-how-many-c) This member function may be overridden by derived classes. It must return a guaranteed lower bound on the number of characters that can be read from the device before uflow or underflow returns EOF. By default 0 is returned (meaning no or some characters are returned before the latter two functions return EOF). When a positive value is returned then the next call of u(nder)flow does not return EOF.

• virtual int uflow():

    This member function may be overridden by derived classes to reload an input buffer with fresh characters. Its default implementation is to call underflow (see below). If underflow() fails, EOF is returned. Otherwise, the next available character is returned as *gptr() following a gbump(-1). Uflow also moves the pending character that is returned to the backup sequence. This is different from underflow(), which merely returns the next available character, but does not alter the input pointer positions.

    When no input buffering is required this function, rather than underflow, can be overridden to produce the next available character from the device to read from.

• virtual int underflow():

    This member function may be overridden by derived classes to read another character from the device. The default implementation is to return EOF.

    It is called when

    - there is no input buffer (eback() == 0)
    - gptr() >= egptr(): the input buffer is exhausted.
Often, when buffering is used, the complete buffer is not refreshed as this would make it impossible to put back characters immediately following a reload. Instead, buffers are often refreshed in halves. This system is called a split buffer.

Classes derived from streambuf for reading normally at least override underflow. The prototypical example of an overridden underflow function looks like this:

```cpp
int underflow()
{
    if (not refillTheBuffer())  // assume a member d_buffer is available
        return EOF;            // reset the input buffer pointers
    setg(d_buffer, d_buffer, d_buffer + d_nCharsRead);
    // return the next available character
    // (the cast is used to prevent
    // misinterpretations of 0xff characters
    // as EOF)
    return static_cast<unsigned char>(*gptr());
}
```

- virtual streamsize xsgetn(char *buffer, streamsize n):

  This member function may be overridden by derived classes to retrieve at once n characters from the input device. The default implementation is to call sbumpc for every single character meaning that by default this member (eventually) calls underflow for every single character. The function returns the actual number of characters read or EOF. Once EOF is returned the streambuf stops reading the device.

### 14.8.1.2 Protected members for output operations

The following protected members are available for output operations. Again, some members may be overridden by derived classes:

- virtual int overflow(int c):

  This member function may be overridden by derived classes to flush the characters currently stored in the output buffer to the output device, and then to reset the output buffer pointers so as to represent an empty buffer. Its parameter c is initialized to the next character to be processed. If no output buffering is used overflow is called for every single character that is written to the streambuf object. No output buffering is accomplished by setting the buffer pointers (using, setp, see below) to 0. The default implementation returns EOF, indicating that no characters can be written to the device.

Classes derived from streambuf for writing normally at least override overflow. The prototypical example of an overridden overflow function looks like this:

```cpp
int OPdStreambuf::overflow(int c)
{
    sync();  // flush the buffer
    if (c != EOF) // write a character?
    {
        *pptr() = static_cast<char>(c);  // put it into the buffer
        pbump(1);                       // advance the buffer's pointer
    }
    return c;
}
```
14.8. THE ‘STREAMBUF’ CLASS

• char *pbase():
  Streambuf maintains three pointers controlling its output buffer: pbase points to the beginning of the output buffer area. See also figure 14.4.

• char *epptr():
  Streambuf maintains three pointers controlling its output buffer: epptr points just beyond the output buffer's last available location. See also figure 14.4. If pptr (see below) equals epptr the buffer must be flushed. This is implemented by calling overflow, see before.

• void pbump(int n):
  The location returned by pptr (see below) is advanced by n. The next character written to the stream will be entered at that location.

• char *pptr():
  Streambuf maintains three pointers controlling its output buffer: pptr points to the location in the output buffer where the next available character should be written. See also figure 14.4.

• void setp(char *beg, char *beyond):
  Streambuf's output buffer is initialized to the locations passed to setp. Beg points to the beginning of the output buffer and beyond points just beyond the last available location of the output buffer. Use setp(0, 0) to indicate that no buffering should be used. In that case overflow is called for every single character to write to the device.

• virtual streamsize xsputn(char const *buffer, streamsize n):
  This member function may be overridden by derived classes to write a series of at most n characters to the output buffer. The actual number of inserted characters is returned. If EOF is returned writing to the device stops. The default implementation calls sputc for each individual character. Redefine this member if, e.g., the streambuf should support the ios::openmode ios::app. Assuming the class MyBuf, derived from streambuf, features a data member ios::openmode d_mode (representing the requested ios::openmode), and a member write(char const *buf, streamsize len) (writing len bytes at pptr()), then the following code acknowledges the ios::app mode:

```cpp
std::streamsize MyStreambuf::xsputn(char const *buf, std::streamsize len) {
    if (d_openMode & ios::app)
        seekoff(0, ios::end);

    return write(buf, len);
}
```

14.8.1.3 Protected members for buffer manipulation

Several protected members are related to buffer management and positioning:
• virtual streambuf *setbuf(char *buffer, streamsize n):
  This member function may be overridden by derived classes to install a buffer. The
default implementation performs no actions. It is called by pubsetbuf.

• virtual ios::pos_type seekoff(ios::off_type offset, ios::seekdir way,
  ios::openmode mode = ios::in |ios::out):
  This member function may be overridden by derived classes to reset the next pointer
  for input or output to a new relative position (using ios::beg, ios::cur or
  ios::end). The default implementation indicates failure by returning -1. The func-
tion is called when tellg or tellp are called. When derived class supports seeking,
then it should also define this function to handle repositioning requests. It is called
by pubseekoff. The new position or an invalid position (i.e., -1) is returned.

• virtual ios::pos_type seekpos(ios::pos_type offset, ios::openmode mode =
  ios::in |ios::out):
  This member function may be overridden by derived classes to reset the next pointer
  for input or output to a new absolute position (i.e, relative to ios::beg). The default
  implementation indicates failure by returning -1.

• virtual int sync():
  This member function may be overridden by derived classes to flush the output buffer
to the output device or to reset the input device just beyond the position of the char-
acter that was returned last. It returns 0 on success, -1 on failure. The default im-
plementation (not using a buffer) is to return 0, indicating successful syncing. This
member is used to ensure that any characters that are still buffered are written to
the device or to put unconsumed characters back to the device when the streambuf
object ceases to exist.

14.8.1.4 Deriving classes from 'streambuf'

When classes are derived from streambuf at least underflow should be overridden by classes
intending to read information from devices, and overflow should be overridden by classes intending
to write information to devices. Several examples of classes derived from streambuf are provided
in chapter 24.

Fstream class type objects use a combined input/output buffer. This is a result from that istream
and ostream being virtually derived from ios, which class contains the streambuf. To construct
a class supporting both input and output using separate buffers, the streambuf itself may define
two buffers. When seekoff is called for reading, a mode parameter can be set to ios::in, other-
wise to ios::out. Thus the derived class knows whether it should access the read buffer or the
write buffer. Of course, underflow and overflow do not have to inspect the mode flag as they by
implication know on which buffer they should operate.

14.8.2 The class ‘filebuf’

The class filebuf is a specialization of streambuf used by the file stream classes. Before using
a filebuf the header file <fstream> must be included.

In addition to the (public) members that are available through the class streambuf, filebuf
offers the following (public) members:
14.9. A POLYMORPHIC EXCEPTION CLASS

- filebuf():
  Filebuf offers a public constructor. It initializes a plain filebuf object that is not yet connected to a stream.

- bool is_open():
  True is returned if the filebuf is actually connected to an open file, false otherwise. See the open member, below.

- filebuf *open(char const *name, ios::openmode mode):
  Associates the filebuf object with a file whose name is provided. The file is opened according to the provided openmode.

- filebuf *close():
  Closes the association between the filebuf object and its file. The association is automatically closed when the filebuf object ceases to exist.

14.9 A polymorphic exception class

Earlier in the C++ Annotations (section 10.3.1) we hinted at the possibility of designing a class Exception whose process member would behave differently, depending on the kind of exception that was thrown. Now that we’ve introduced polymorphism we can further develop this example.

It probably does not come as a surprise that our class Exception should be a polymorphic base class from which special exception handling classes can be derived. In section 10.3.1 a member severity was used offering functionality that may be replaced by members of the Exception base class.

The base class Exception may be designed as follows:

```cpp
#include <iostream>
#include <string>

class Exception
{
  std::string d_reason;

public:
  Exception(std::string const &reason);
  virtual ~Exception();
  std::ostream &insertInto(std::ostream &out) const;
  void handle() const;

private:
  virtual void action() const;
};

inline void Exception::action() const
{
  throw;
}
CHAPTER 14. POLYMORPHISM

Objects of this class may be inserted into `ostream`s but the core element of this class is the virtual member function `action`, by default rethrowing an exception.

A derived class `Warning` simply prefixes the thrown warning text by the text `Warning:`, but a derived class `Fatal` overrides `Exception::action` by calling `std::terminate`, forcefully terminating the program.

Here are the classes `Warning` and `Fatal`

```cpp
#ifndef WARNINGEXCEPTION_H_
#define WARNINGEXCEPTION_H_

#include "exception.h"

class Warning: public Exception
{
    public:
        Warning(std::string const &reason)
        :
            Exception("Warning: " + reason)
        {}
};
#endif

#ifndef FATAL_H_
#define FATAL_H_

#include "exception.h"

class Fatal: public Exception
{
    public:
        Fatal(std::string const &reason);
};
#endif
```
When the example program is started without arguments it throws a Fatal exception, otherwise it throws a Warning exception. Of course, additional exception types could also easily be defined. To make the example compilable the Exception destructor is defined above main. The default destructor cannot be used, as it is a virtual destructor. In practice the destructor should be defined in its own little source file:

```cpp
#include "warning.h"
#include "fatal.h"

Exception::~Exception() {}

using namespace std;

int main(int argc, char **argv)
try {
    try {
        if (argc == 1)
            throw Fatal("Missing Argument");
        else
            throw Warning("the argument is ignored");
    } catch (Exception const &e) {
        cout << e << '\n';
        e.handle();
    }
} catch(...) {
    cout << "caught rethrown exception\n";
}
```
14.10 How polymorphism is implemented

This section briefly describes how polymorphism is implemented in C++. It is not necessary to understand how polymorphism is implemented if you just want to use polymorphism. However, we think it’s nice to know how polymorphism is possible. Also, knowing how polymorphism is implemented clarifies why there is a (small) penalty to using polymorphism in terms of memory usage and efficiency.

The fundamental idea behind polymorphism is that the compiler does not know which function to call at compile-time. The appropriate function is selected at run-time. That means that the address of the function must be available somewhere, to be looked up prior to the actual call. This ‘somewhere’ place must be accessible to the object in question. So when a Vehicle *vp points to a Truck object, then vp->mass() calls Truck’s member function. The address of this function is obtained through the actual object to which vp points.

Polymorphism is commonly implemented as follows: an object containing virtual member functions also contains, usually as its first data member a hidden data member, pointing to an array containing the addresses of the class’s virtual member functions. The hidden data member is usually called the vpointer, the array of virtual member function addresses the vtable.

The class’s vtable is shared by all objects of that class. The overhead of polymorphism in terms of memory consumption is therefore:

- one vpointer data member per object pointing to:
- one vtable per class.

Consequently, a statement like vp->mass first inspects the hidden data member of the object pointed to by vp. In the case of the vehicle classification system, this data member points to a table containing two addresses: one pointer to the function mass and one pointer to the function setMass (three pointers if the class also defines (as it should) a virtual destructor). The actually called function is determined from this table.

The internal organization of the objects having virtual functions is illustrated in figures Figure 14.5 and Figure 14.6 (originals provided by Guillaume Caumon).3

As shown by figures Figure 14.5 and Figure 14.6, objects potentially using virtual member functions must have one (hidden) data member to address a table of function pointers. The objects of the

3mailto:Guillaume.Caumon@ensg.inpl-nancy.fr
classes Vehicle and Car both address the same table. The class Truck, however, overrides mass. Consequently, Truck needs its own vtable.

A small complication arises when a class is derived from multiple base classes, each defining virtual functions. Consider the following example:

```cpp
class Base1
{
public:
    virtual ~Base1();
    void fun1(); // calls vOne and vTwo
private:
    virtual void vOne();
    virtual void vTwo();
};
class Base2
{
public:
    virtual ~Base2();
    void fun2(); // calls vThree
private:
    virtual void vThree();
};
class Derived: public Base1, public Base2
{
public:
    virtual ~Derived();
private:
    virtual ~vOne();
    virtual ~vThree();
};
```

In the example Derived is multiply derived from Base1 and Base2, each supporting virtual functions. Because of this, Derived also has virtual functions, and so Derived has a vtable allowing
a base class pointer or reference to access the proper virtual member.

When Derived::fun1 is called (or a Base1 pointer pointing to fun1 calls fun1) then fun1 calls Derived::vOne and Base1::vTwo. Likewise, when Derived::fun2 is called Derived::vThree is called.

The complication occurs with Derived’s vtable. When fun1 is called its class type determines the vtable to use and hence which virtual member to call. So when vOne is called from fun1, it is presumably the second entry in Derived’s vtable, as it must match the second entry in Base1’s vtable. However, when fun2 calls vThree it apparently is also the second entry in Derived’s vtable as it must match the second entry in Base2’s vtable.

Of course this cannot be realized by a single vtable. Therefore, when multiple inheritance is used (each base class defining virtual members) another approach is followed to determine which virtual function to call. In this situation (cf. figure Figure 14.7) the class Derived receives two vtables, one for each of its base classes and each Derived class object harbors two hidden vpointers, each one pointing to its corresponding vtable.

Since base class pointers, base class references, or base class interface members unambiguously refer to one of the base classes the compiler can determine which vpointer to use.

The following therefore holds true for classes multiply derived from base classes offering virtual member functions:
the derived class defines a vtable for each of its base classes offering virtual members;

• Each derived class object contains as many hidden vpointers as it has vtables.

• Each of a derived class object’s vpointers points to a unique vtable and the vpointer to use is determined by the class type of the base class pointer, the base class reference, or the base class interface function that is used.

14.11 Undefined reference to vtable ...

Occasionally, the linker generates an error like the following:

In function 'Derived::Derived()':
  : undefined reference to 'vtable for Derived'

This error is generated when a virtual function's implementation is missing in a derived class, but the function is mentioned in the derived class's interface.

Such a situation is easily encountered:

• Construct a (complete) base class defining a virtual member function;

• Construct a Derived class mentioning the virtual function in its interface;

• The Derived class's virtual function is not implemented. Of course, the compiler doesn't know that the derived class's function is not implemented and will, when asked, generate code to create a derived class object;

• Eventually, the linker is unable to find the derived class's virtual member function. Therefore, it is unable to construct the derived class's vtable;

• The linker complains with the message:

  undefined reference to 'vtable for Derived'

Here is an example producing the error:

class Base
{
    virtual void member();
};
inline void Base::member()
{}
class Derived: public Base
{
    virtual void member();       // only declared
};
int main()
{
    Derived d;                  // Will compile, since all members were declared.
    // Linking will fail, since we don’t have the
    // implementation of Derived::member()
}
It's of course easy to correct the error: implement the derived class's missing virtual member function.

Virtual functions should never be implemented inline. Since the vtable contains the addresses of the class's virtual functions, these functions must have addresses and so they must have been compiled as real (out-of-line) functions. By defining virtual functions inline you run the risk that the compiler simply overlooks those functions as they may very well never be explicitly called (but only polymorphically, from a base class pointer or reference). As a result their addresses may never enter their class's vtables (and even the vtable itself might remain undefined), causing linkage problems or resulting in programs showing unexpected behavior. All these kinds of problems are simply avoided: never define virtual members inline (see also section 7.9.2.1).

14.12 Virtual constructors

In section 14.2 we learned that C++ supports virtual destructors. Like many other object oriented languages (e.g., Java), however, the notion of a virtual constructor is not supported. Not having virtual constructors becomes a liability when only base class references or pointers are available, and a copy of a derived class object is required. Gamma et al. (1995) discuss the Prototype design pattern to deal with this situation.

According to the Prototype Design Pattern each derived class is given the responsibility of implementing a member function returning a pointer to a copy of the object for which the member is called. The usual name for this function is clone. Separating the user interface from the reimplementation interface clone is made part of the interface and newCopy is defined in the reimplementation interface. A base class supporting 'cloning' defines a virtual destructor, clone, returning newCopy's return value and the virtual copy constructor, a pure virtual function, having the prototype virtual Base *newCopy() const = 0. As newCopy is a pure virtual function all derived classes must now implement their own 'virtual constructor'.

This setup suffices in most situations where we have a pointer or reference to a base class, but it fails when used with abstract containers. We can’t create a vector<Base>, with Base featuring the pure virtual copy member in its interface, as Base is called to initialize new elements of such a vector. This is impossible as newCopy is a pure virtual function, so a Base object can’t be constructed.

The intuitive solution, providing newCopy with a default implementation, defining it as an ordinary virtual function, fails too as the container calls Base(Base const &other), which would have to call newCopy to copy other. At this point it is unclear what to do with that copy, as the new Base object already exists, and contains no Base pointer or reference data member to assign newCopy’s return value to.

Alternatively (and preferred) the original Base class (defined as an abstract base class) is kept as-is and a wrapper class Clonable is used to manage the Base class pointers returned by newCopy. In chapter 17 ways to merge Base and Clonable into one class are discussed, but for now we’ll define Base and Clonable as separate classes.

The class Clonable is a very standard class. It contains a pointer member so it needs a copy constructor, destructor, and overloaded assignment operator. It’s given at least one non-standard member: Base &base() const, returning a reference to the derived object to which Clonable’s Base * data member refers. It is also provided with an additional constructor to initialize its Base * data member.

Any non-abstract class derived from Base must implement Base *newCopy(), returning a pointer to a newly created (allocated) copy of the object for which newCopy is called.
14.12. VIRTUAL CONSTRUCTORS

Once we have defined a derived class (e.g., Derived1), we can put our Clonable and Base facilities to good use. In the next example we see main defining a vector<Clonable>. An anonymous Derived1 object is then inserted into the vector using the following steps:

- A new anonymous Derived1 object is created;
- It initializes a Clonable using Clonable(Base *bp);
- The just created Clonable object is inserted into the vector, using Clonable's move constructor. There are only temporary Derived and Clonable objects at this point, so no copy construction is required.

In this sequence, only the Clonable object containing the Derived1 * is used. No additional copies need to be made (or destroyed).

Next, the base member is used in combination with typeid to show the actual type of the Base & object: a Derived1 object.

Main then contains the interesting definition vector<Clonable> v2(bv). Here a copy of bv is created. This copy construction observes the actual types of the Base references, making sure that the appropriate types appear in the vector's copy.

At the end of the program, we have created two Derived1 objects, which are correctly deleted by the vector's destructors. Here is the full program, illustrating the 'virtual constructor' concept:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <typeinfo>

// Base and its inline member:
class Base
{
    public:
        virtual ~Base();
        Base *clone() const;
    private:
        virtual Base *newCopy() const = 0;
    }
inline Base *Base::clone() const
{
    return newCopy();
}

// Clonable and its inline members:
class Clonable
{
    Base *d_bp;

    public:
        Clonable();
        explicit Clonable(Base *base);
        ~Clonable();

4 Jesse van den Kieboom created an alternative implementation of a class Clonable, implemented as a class template. His implementation is found here. 
```
Clonable(const Clonable &other);
Clonable(Clonable &&tmp);
Clonable &operator=(const Clonable &other);
Clonable &operator=(Clonable &&tmp);

Base &base() const;

inline Clonable::Clonable():
    d_bp(0)
{
}
inline Clonable::Clonable(Base *bp):
    d_bp(bp)
{
}
inline Clonable::Clonable(const Clonable &other):
    d_bp(other.d_bp->clone())
{
}
inline Clonable::Clonable(Clonable &&tmp):
    d_bp(tmp.d_bp)
{
    tmp.d_bp = 0;
}
inline Clonable::~Clonable()
{
    delete d_bp;
}
inline Base &Clonable::base() const
{
    return *d_bp;
}

// Derived and its inline member:
class Derived1: public Base
{
    public:
        ~Derived1();
    private:
        virtual Base *newCopy() const;
    inline Base *Derived1::newCopy() const
        { return new Derived1(*this); }
};

// Members not implemented inline:
Base::~Base()
{};
Clonable &Clonable::operator=(Clonable const &other)
{
    Clonable tmp(other);
    std::swap(d_bp, tmp.d_bp);
return *this;
}
Clonable &Clonable::operator=(Clonable &&tmp)
{
    std::swap(d_bp, tmp.d_bp);
    return *this;
}

Derived1::~Derived1()
{
    std::cout << "~Derived1() called\n"
}
// The main function:
using namespace std;

int main()
{
    vector<Clonable> bv;
    bv.push_back(CIonable(new Derived1()));
    cout << "bv[0].name: " << typeid(bv[0].base()).name() << '\n';

    vector<Clonable> v2(bv);
    cout << "v2[0].name: " << typeid(v2[0].base()).name() << '\n';
}
/*
Output:
  bv[0].name: Derived1
  v2[0].name: Derived1
~Derived1() called
~Derived1() called
*/
Chapter 15

Friends

In all examples discussed up to now, we've seen that private members are only accessible by the members of their class. This is good, as it enforces encapsulation and data hiding. By encapsulating functionality within a class we prevent that a class exposes multiple responsibilities; by hiding data we promote a class's data integrity and we prevent that other parts of the software become implementation dependent on the data that belong to a class.

In this (very) short chapter we introduce the friend keyword and the principles that underly its use. The bottom line being that by using the friend keyword functions are granted access to a class's private members. Even so, this does not imply that the principle of data hiding is abandoned when the friend keyword is used.

In this chapter the topic of friendship among classes is not discussed. Situations in which it is natural to use friendship among classes are discussed in chapters 17 and 21 and such situations are natural extensions of the way friendship is handled for functions.

There should be a well-defined conceptual reason for declaring friendship (i.e., using the friend keyword). The traditionally offered definition of the class concept usually looks something like this:

A class is a set of data together with the functions that operate on that set of data.

As we've seen in chapter 11 some functions have to be defined outside of a class interface. They are defined outside of the class interface to allow promotions for their operands or to extend the facilities of existing classes not directly under our control. According to the above traditional definition of the class concept those functions that cannot be defined in the class interface itself should nevertheless be considered functions belonging to the class. Stated otherwise: if permitted by the language’s syntax they would certainly have been defined inside the class interface. There are two ways to implement such functions. One way consists of implementing those functions using available public member functions. This approach was used, e.g., in section 11.2. Another approach applies the definition of the class concept to those functions. By stating that those functions in fact belong to the class they should be given direct access to the data members of objects. This is accomplished by the friend keyword.

As a general principle we state that all functions operating on the data of objects of a class that are declared in the same file as the class interface itself belong to that class and may be granted direct access to the class's data members.
CHAPTER 15. FRIENDS

15.1 Friend functions

In section 11.2 the insertion operator of the class Person (cf. section 9.3) was implemented like this:

```cpp
ostream &operator<<(ostream &out, Person const &person)
{
    return
    out <<
    "Name: " << person.name() << ", "
    "Address: " << person.address() << ", "
    "Phone: " << person.phone();
}
```

Person objects can now be inserted into streams.

However, this implementation required three member functions to be called, which may be considered a source of inefficiency. An improvement would be reached by defining a member Person::insertInto and let operator« call that function. These two functions could be defined as follows:

```cpp
std::ostream &operator<<(std::ostream &out, Person const &person)
{
    return person.insertInto(out);
}
std::ostream &Person::insertInto(std::ostream &out)
{
    return
    out << "Name: " << d_name << ", "
    "Address: " << d_address << ", "
    "Phone: " << d_phone;
}
```

As insertInto is a member function it has direct access to the object’s data members so no additional member functions must be called when inserting person into out.

The next step consists of realizing that insertInto is only defined for the benefit of operator«, and that operator«, as it is declared in the header file containing Person’s class interface should be considered a function belonging to the class Person. The member insertInto can therefore be omitted when operator« is declared as a friend.

Friend functions must be declared as friends in the class interface. These friend declarations are not member functions, and so they are independent of the class’s private, protected and public sections. Friend declaration may be placed anywhere in the class interface. Convention dictates that friend declarations are listed directly at the top of the class interface. The class Person, using friend declaration for its extraction and insertion operators starts like this:

```cpp
class Person
{
    friend std::ostream &operator<<(std::ostream &out, Person &pd);
    friend std::istream &operator>>(std::istream &in, Person &pd);

    // previously shown interface (data and functions)
};
```
The insertion operator may now directly access a `Person` object's data members:

```cpp
std::ostream &operator<<(std::ostream &out, Person const &person)
{
    return cout << "Name: " << person.d_name << ", "
             << "Address: " << person.d_address << ", "
             << "Phone: " << person.d_phone;
}
```

Friend declarations are true declarations. Once a class contains friend declarations these friend functions do not have to be declared again below the class's interface. This also clearly indicates the class designer's intent: the friend functions are declared by the class, and can thus be considered functions belonging to the class.

### 15.2 Extended friend declarations

C++11 simplifies friend declarations by adding extended friend declarations to the language. When a class is declared as a friend, then the `class` keyword no longer has to be provided. E.g.,

```cpp
class Friend; // declare a class
typedef Friend FriendType; // and a typedef for it
using FName = Friend; // and a using declaration

class Class1
{
    friend Friend; // FriendType and FName: also OK
};
```

In the pre-C++11 standards the friend declaration required an explicit `class`, e.g., `friend class Friend`.

The explicit use of `class` remains required if the compiler hasn't seen the friend’s name yet. E.g.,

```cpp
class Class1
{
    // friend Unseen; // fails to compile: Unseen unknown.
    friend class Unseen; // OK
};
```

Section 22.10 covers the use of extended friend declarations in class templates.
Chapter 16

Classes Having Pointers To Members

Classes having pointer data members have been discussed in detail in chapter 9. Classes defining pointer data-members deserve some special attention, as they usually require the definitions of copy constructors, overloaded assignment operators and destructors.

Situations exist where we do not need a pointer to an object but rather a pointer to members of a class. Pointers to members can profitably be used to configure the behavior of objects of classes. Depending on which member a pointer to a member points to objects will show certain behavior.

Although pointers to members have their use, polymorphism can frequently be used to realize comparable behavior. Consider a class having a member process performing one of a series of alternate behaviors. Instead of selecting the behavior of choice at object construction time the class could use the interface of some (abstract) base class, passing an object of some derived class to its constructor and could thus configure its behavior. This allows for easy, extensible and flexible configuration, but access to the class's data members would be less flexible and would possibly require the use of 'friend' declarations. In such cases pointers to members may actually be preferred as this allows for (somewhat less flexible) configuration as well as direct access to a class's data members.

So the choice apparently is between on the one hand ease of configuration and on the other hand ease of access to a class's data members. In this chapter we'll concentrate on pointers to members, investigating what these pointers have to offer.

16.1 Pointers to members: an example

Knowing how pointers to variables and objects are used does not intuitively lead to the concept of pointers to members. Even if the return types and parameter types of member functions are taken into account, surprises can easily be encountered. For example, consider the following class:

```cpp
class String
{
    char const *(*d_sp)() const;

public:
    char const *get() const;
};
```
For this class, it is not possible to let $\text{char const \ast(*d\_sp)() const}$ point to the `String::get` member function as `d\_sp` cannot be given the address of the member function `get`.

One of the reasons why this doesn’t work is that the variable `d\_sp` has global scope (it is a pointer to a function, not a pointer to a function within `String`), while the member function `get` is defined within the `String` class, and thus has class scope. The fact that `d\_sp` is a data member of the class `String` is irrelevant here. According to `d\_sp`’s definition, it points to a function living somewhere outside of the class.

Consequently, to define a pointer to a member (either data or function, but usually a function) of a class, the scope of the pointer must indicate class scope. Doing so, a pointer to the member `String::get` is defined like this:

$$\text{char const \ast(String::\ast d\_sp)() const;}$$

So, by prefixing the `\ast d\_sp` pointer data member by `String::`, it is defined as a pointer in the context of the class `String`. According to its definition it is a pointer to a function in the class `String`, not expecting arguments, not modifying its object’s data, and returning a pointer to constant characters.

### 16.2 Defining pointers to members

Pointers to members are defined by prefixing the normal pointer notation with the appropriate class plus scope resolution operator. Therefore, in the previous section, we used $\text{char const \ast(String::\ast d\_sp)() const}$ to indicate that `d\_sp`

- is a pointer (`\ast d\_sp`);
- points to something in the class `String (String::\ast d\_sp)`;
- is a pointer to a const function, returning a `char const \ast (char const \ast(String::\ast d\_sp)() const)`.

The prototype of a matching function is therefore:

$$\text{char const \ast String::somefun() const;}$$

which is any `const` parameterless function in the class `String`, returning a `char const \ast`.

When defining pointers to members the standard procedure for constructing pointers to functions can still be applied:

- put parentheses around the fully qualified function name (i.e., the function’s header, including the function’s class name):

  $$\text{char const \ast ( String::somefun ) () const}$$

- Put a pointer (a star (`\ast`)) character immediately before the function name itself:

  $$\text{char const \ast ( String:: \ast somefun ) () const}$$
16.2. DEFINING POINTERS TO MEMBERS

• Replace the function name with the name of the pointer variable:

\[ \text{char const } * \text{ (String::*d_sp) () const} \]

Here is another example, defining a pointer to a data member. Assume the class \text{String} contains a \text{string d_text} member. How to construct a pointer to this member? Again we follow standard procedure:

• put parentheses around the fully qualified variable name:

\[ \text{std::string (String::d_text)} \]

• Put a pointer (a star (\*) character immediately before the variable-name itself:

\[ \text{std::string (String::*d_text)} \]

• Replace the variable name with the name of the pointer variable:

\[ \text{std::string (String::*tp)} \]

In this case, the parentheses are superfluous and may be omitted:

\[ \text{std::string (String::*tp)} \]

Alternatively, a very simple rule of thumb is

• Define a normal (i.e., global) pointer variable,
• Prefix the class name to the pointer character, once you point to something inside a class

For example, the following pointer to a global function

\[ \text{char const } * \text{ (*sp) () const;} \]

becomes a pointer to a member function after prefixing the class-scope:

\[ \text{char const } * \text{ (String::*sp) () const;} \]

Nothing forces us to define pointers to members in their target \text{String} classes. Pointers to members \text{may} be defined in their target classes (so they become data members), or in another class, or as a local variable or as a global variable. In all these cases the pointer to member variable can be given the address of the kind of member it points to. The important part is that a pointer to member can be initialized or assigned without requiring the existence an object of the pointer's target class.

Initializing or assigning an address to such a pointer merely indicates to which member the pointer points. This can be considered some kind of \text{relative address}; relative to the object for which the function is called. No object is required when pointers to members are initialized or assigned. While it is allowed to initialize or assign a pointer to member, it is (of course) not possible to \text{call} those members without specifying an object of the correct type.

In the following example initialization of and assignment to pointers to members is illustrated (for illustration purposes all members of the class \text{PointerDemo} are defined \text{public}). In the example
itself the &-operator is used to determine the addresses of the members. These operators as well as
the class-scopes are required. Even when used inside member implementations:

```cpp
#include <stddef>

class PointerDemo
{
    public:
        size_t d_value;
        size_t get() const;
};

inline size_t PointerDemo::get() const
{
    return d_value;
}

int main()
{
    // initialization
    size_t (PointerDemo::*getPtr)() const = &PointerDemo::get;
    size_t PointerDemo::*valuePtr = &PointerDemo::d_value;
    getPtr = &PointerDemo::get; // assignment
    valuePtr = &PointerDemo::d_value;
}
```

This involves nothing special. The difference with pointers at global scope is that we’re now re-
stricting ourselves to the scope of the PointerDemo class. Because of this restriction, all pointer
definitions and all variables whose addresses are used must be given the PointerDemo class scope.

Pointers to members can also be used with virtual member functions. No special syntax is re-
quired when pointing to virtual members. Pointer construction, initialization and assignment is
done identically to the way it is done with non-virtual members.

### 16.3 Using pointers to members

Using pointers to members to call a member function requires the existence of an object of the class
of the members to which the pointer to member refers to. With pointers operating at global scope,
the dereferencing operator * is used. With pointers to objects the field selector operator operating on
pointers (->) or the field selector operating operating on objects (.) can be used to select appropriate
members.

To use a pointer to member in combination with an object the pointer to member field selector (.)
must be specified. To use a pointer to a member via a pointer to an object the ‘pointer to member
field selector through a pointer to an object’ (->_) must be specified. These two operators combine
the notions of a field selection (the . and -> parts) to reach the appropriate field in an object and
of dereferencing: a dereference operation is used to reach the function or variable the pointer to
member points to.

Using the example from the previous section, let’s see how we can use pointers to member functions
and pointers to data members:

```cpp
#include <iostream>
```
16.3. USING POINTERS TO MEMBERS

class PointerDemo
{
    public:
        size_t d_value;
        size_t get() const;
};

inline size_t PointerDemo::get() const
{
    return d_value;
}

using namespace std;

int main()
{
    // initialization
    size_t (PointerDemo::*getPtr)() const = &PointerDemo::get;
    size_t PointerDemo::*valuePtr = &PointerDemo::d_value;

    PointerDemo object;  // (1) (see text)
    PointerDemo *ptr = &object;

    object.*valuePtr = 12345;  // (2)
    cout << object.*valuePtr << 'n' <<
        object.d_value << 'n';

    ptr->*valuePtr = 54321;  // (3)
    cout << object.d_value << 'n' <<
        (object.*getPtr)() << 'n' <<
        (ptr->*getPtr)() << 'n';
}

We note:

- At (1) a PointerDemo object and a pointer to such an object is defined.
- At (2) we specify an object (and hence the .* operator) to reach the member valuePtr points to. This member is given a value.
- At (3) the same member is assigned another value, but this time using the pointer to a PointerDemo object. Hence we use the ->* operator.
- At (4) the .* and ->* are used once again, this time to call a function through a pointer to member. As the function argument list has a higher priority than the pointer to member field selector operator, the latter must be protected by parentheses.

Pointers to members can be used profitably in situations where a class has a member that behaves differently depending on a configuration setting. Consider once again the class Person from section 9.3. Person defines data members holding a person's name, address and phone number. Assume we want to construct a Person database of employees. The employee database can be queried, but depending on the kind of person querying the database either the name, the name and phone number or all stored information about the person is made available. This implies that a member function like address must return something like 'not available' in cases where the person...
querying the database is not allowed to see the person’s address, and the actual address in other cases.

The employee database is opened specifying an argument reflecting the status of the employee who wants to make some queries. The status could reflect his or her position in the organization, like BOARD, SUPERVISOR, SALESPERSON, or CLERK. The first two categories are allowed to see all information about the employees, a SALESPERSON is allowed to see the employee’s phone numbers, while the CLERK is only allowed to verify whether a person is actually a member of the organization.

We now construct a member string personInfo(char const *name) in the database class. A standard implementation of this class could be:

```cpp
string PersonData::personInfo(char const *name)
{
    Person *p = lookup(name); // see if ‘name’ exists
    if (!p)
        return "not found";
    switch (d_category)
    {
        case BOARD:
        case SUPERVISOR:
            return allInfo(p);
        case SALESPERSON:
            return noPhone(p);
        case CLERK:
            return nameOnly(p);
    }
}
```

Although it doesn’t take much time, the switch must nonetheless be evaluated every time personInfo is called. Instead of using a switch, we could define a member d_infoPtr as a pointer to a member function of the class PersonData returning a string and expecting a pointer to a Person as its argument.

Instead of evaluating the switch this pointer can be used to point to allInfo, noPhone or nameOnly. Furthermore, the member function the pointer points to will be known by the time the PersonData object is constructed and so its value needs to be determined only once (at the PersonData object’s construction time).

Having initialized d_infoPtr the personInfo member function is now implemented simply as:

```cpp
string PersonData::personInfo(char const *name)
{
    Person *p = lookup(name); // see if ‘name’ exists

    return p ? (this->*d_infoPtr)(p) : "not found";
}
```

The member d_infoPtr is defined as follows (within the class PersonData, omitting other members):

```cpp
class PersonData
```
16.4 Pointers to static members

Static members of a class can be used without having available an object of their class. Public static members can be called like free functions, albeit that their class names must be specified when they are called.

Assume a class String has a public static member function count, returning the number of string objects created so far. Then, without using any String object the function String::count may be called:

```cpp
void fun()
{
    cout << String::count() << '
';
}
```

Public static members can be called like free functions (but see also section 8.2.1). Private static members can only be called within the context of their class, by their class's member or friend functions.

Since static members have no associated objects their addresses can be stored in ordinary function pointer variables, operating at the global level. Pointers to members cannot be used to store addresses of static members. Example:

```cpp
{
    string (PersonData::*d_infoPtr)(Person *p);
}
```

Finally, the constructor initializes d_infoPtr. This could be realized using a simple switch:

```cpp
PersonData::PersonData(PersonData::EmployeeCategory cat)
:
    switch (cat)
    {
    case BOARD:
        case SUPERVISOR:
            d_infoPtr = &PersonData::allInfo;
            break;
    case SALESPERSON:
            d_infoPtr = &PersonData::noPhone;
            break;
    case CLERK:
            d_infoPtr = &PersonData::nameOnly;
            break;
    }
```

Note how addresses of member functions are determined. The class PersonData scope must be specified, even though we’re already inside a member function of the class PersonData.

An example using pointers to data members is provided in section 19.1.60, in the context of the stable_sort generic algorithm.
void fun()
{
    size_t (*pf)() = String::count;
    // initialize pf with the address of a static member function
    cout << (*pf)() << '\n';
    // displays the value returned by String::count()
}

16.5 Pointer sizes

An interesting characteristic of pointers to members is that their sizes differ from those of 'normal' pointers. Consider the following little program:

```cpp
#include <string>
#include <iostream>

class X
{
    public:
        void fun();
        std::string d_str;
};
inline void X::fun()
{
    std::cout << "hello\n";
}

using namespace std;
int main()
{
    cout <<
        "size of pointer to data-member: " << sizeof(&X::d_str) << \n"
        "size of pointer to member function: " << sizeof(&X::fun) << \n"
        "size of pointer to non-member data: " << sizeof(char *) << \n"
        "size of pointer to free function: " << sizeof(&printf) << '\n';
}
/*
generated output (on 32-bit architectures):
size of pointer to data-member: 4
size of pointer to member function: 8
size of pointer to non-member data: 4
size of pointer to free function: 4
*/

On a 32-bit architecture a pointer to a member function requires eight bytes, whereas other kind of pointers require four bytes (Using Gnu’s g++ compiler).

Pointer sizes are hardly ever explicitly used, but their sizes may cause confusion in statements like:
printf("%p", &X::fun);

Of course, printf is likely not the right tool to produce the value of these C++ specific pointers. The values of these pointers can be inserted into streams when a union, reinterpreting the 8-byte pointers as a series of size_t char values, is used:

```cpp
#include <string>
#include <iostream>
#include <iomanip>

class X
{
    public:
        void fun();
        std::string d_str;
    }
inline void X::fun()
{
    std::cout << "hello\n";
}

using namespace std;
int main()
{
    union
    {
        void (X::*f)();
        unsigned char *cp;
    }
    u = { &X::fun };

    cout.fill('0');
    cout << hex;
    for (unsigned idx = sizeof(void (X::*))(); idx-- > 0; )
    {
        cout << setw(2) << static_cast<unsigned>(u.cp[idx]);
        cout << '\n';
    }
}
Chapter 17

Nested Classes

Classes can be defined inside other classes. Classes that are defined inside other classes are called nested classes. Nested classes are used in situations where the nested class has a close conceptual relationship to its surrounding class. For example, with the class `string::iterator` is available which provides all characters that are stored in the `string`. This `string::iterator` type could be defined as an object `iterator`, defined as nested class in the class `string`.

A class can be nested in every part of the surrounding class: in the public, protected or private section. Such a nested class can be considered a member of the surrounding class. The normal access and rules in classes apply to nested classes. If a class is nested in the public section of a class, it is visible outside the surrounding class. If it is nested in the protected section it is visible in subclasses, derived from the surrounding class, if it is nested in the private section, it is only visible for the members of the surrounding class.

The surrounding class has no special privileges towards the nested class. The nested class has full control over the accessibility of its members by the surrounding class. For example, consider the following class definition:

```cpp
class Surround
{
    public:
        class FirstWithin
        {
            int d_variable;

            public:
                FirstWithin();
                int var() const;
        }

    private:
        class SecondWithin
        {
            int d_variable;

            public:
                SecondWithin();
                int var() const;
        }
};
```

441
inline int Surround::FirstWithin::var() const
{
    return d_variable;
}
inline int Surround::SecondWithin::var() const
{
    return d_variable;
}

Here access to the members is defined as follows:

- The class FirstWithin is visible outside and inside Surround. The class FirstWithin thus has global visibility.
- FirstWithin's constructor and its member function var are also globally visible.
- The data member d_variable is only visible to the members of the class FirstWithin. Neither the members of Surround nor the members of SecondWithin can directly access FirstWithin::d_variable.
- The class SecondWithin is only visible inside Surround. The public members of the class SecondWithin can also be used by the members of the class FirstWithin, as nested classes can be considered members of their surrounding class.
- SecondWithin's constructor and its member function var also can only be reached by the members of Surround (and by the members of its nested classes).
- SecondWithin::d_variable is only visible to SecondWithin's members. Neither the members of Surround nor the members of FirstWithin can access d_variable of the class SecondWithin directly.
- As always, an object of the class type is required before its members can be called. This also holds true for nested classes.

To grant the surrounding class access rights to the private members of its nested classes or to grant nested classes access rights to the private members of the surrounding class, the classes can be defined as friend classes (see section 17.3).

Nested classes can be considered members of the surrounding class, but members of nested classes are not members of the surrounding class. So, a member of the class Surround may not access FirstWithin::var directly. This is understandable considering that a Surround object is not also a FirstWithin or SecondWithin object. In fact, nested classes are just typenames. It is not implied that objects of such classes automatically exist in the surrounding class. If a member of the surrounding class should use a (non-static) member of a nested class then the surrounding class must define a nested class object, which can thereupon be used by the members of the surrounding class to use members of the nested class.

For example, in the following class definition there is a surrounding class Outer and a nested class Inner. The class Outer contains a member function caller. The member function caller uses the d_inner object that is composed within Outer to call Inner::infunction:

```cpp
class Outer
{
    public:
        void caller();
```
17.1 Defining nested class members

Member functions of nested classes may be defined as inline functions. Inline member functions can be defined as if they were defined outside of the class definition. To define the member function Outer::caller outside of the class Outer, the function's fully qualified name (starting from the outermost class scope (Outer)) must be provided to the compiler. Inline and in-class functions can be defined accordingly. They can be defined and they can use any nested class. Even if the nested class's definition appears later in the outer class's interface.

When (nested) member functions are defined inline, their definitions should be put below their class interface. Static nested data members are also usually defined outside of their classes. If the class FirstWithin would have had a static size_t datamember epoch, it could have been initialized as follows:

```cpp
size_t Surround::FirstWithin::epoch = 1970;
```

Furthermore, multiple scope resolution operators are needed to refer to public static members in code outside of the surrounding class:

```cpp
void showEpoch()
{
    cout << Surround::FirstWithin::epoch;
}
```

Within the class Surround only the FirstWithin:: scope must be used; within the class FirstWithin there is no need to refer explicitly to the scope.

What about the members of the class SecondWithin? The classes FirstWithin and SecondWithin are both nested within Surround, and can be considered members of the surrounding class. Since members of a class may directly refer to each other, members of the class SecondWithin can refer to (public) members of the class FirstWithin. Consequently, members of the class SecondWithin could refer to the epoch member of FirstWithin as FirstWithin::epoch.
17.2 Declaring nested classes

Nested classes may be declared before they are actually defined in a surrounding class. Such forward declarations are required if a class contains multiple nested classes, and the nested classes contain pointers, references, parameters or return values to objects of the other nested classes.

For example, the following class `Outer` contains two nested classes `Inner1` and `Inner2`. The class `Inner1` contains a pointer to `Inner2` objects, and `Inner2` contains a pointer to `Inner1` objects. Cross references require forward declarations. Forward declarations must be given an access specification that is identical to the access specification of their definitions. In the following example the `Inner2` forward declaration must be given in a `private` section, as its definition is also part of the class `Outer`'s private interface:

```cpp
class Outer
{
    private:
        class Inner2; // forward declaration
    class Inner1
    {
        Inner2 *pi2; // points to Inner2 objects
    }
    class Inner2
    {
        Inner1 *pi1; // points to Inner1 objects
    }
};
```

17.3 Accessing private members in nested classes

To grant nested classes access rights to the private members of other nested classes, or to grant a surrounding class access to the private members of its nested classes the `friend` keyword must be used.

Note that no friend declaration is required to grant a nested class access to the private members of its surrounding class. After all, a nested class is a type defined by its surrounding class and as such objects of the nested class are members of the outer class and thus can access all the outer class's members. Here is an example showing this principle. The example won't compile as members of the class `Extern` are denied access to `Outer`'s private members, but `Outer::Inner`'s members can access `Outer`'s private members:

```cpp
class Outer
{
    int d_value;
    static int s_value;

    public:
    Outer()
    :
        d_value(12)
    {}
    class Inner
```
17.3. ACCESSING PRIVATE MEMBERS IN NESTED CLASSES

```cpp
{    
    public:
        Inner()
        {
            cout << "Outer’s static value: " << s_value << ‘\n’;
        }
        Inner(Outer &outer)
        {
            cout << "Outer’s value: " << outer.d_value << ‘\n’;
        }
    
    class Extern // won’t compile!
        {
            public:
                Extern(Outer &outer)
                {
                    cout << "Outer’s value: " << outer.d_value << ‘\n’;
                }
                Extern()
                {
                    cout << "Outer’s static value: " << Outer::s_value << ‘\n’;
                }
        
        int Outer::s_value = 123;
        int main()
        {
            Outer outer;
            Outer::Inner in1;
            Outer::Inner in2(outer);
        }
    
Now consider the situation where a class Surround has two nested classes FirstWithin and SecondWithin. Each of the three classes has a static data member int s_variable:

    class Surround
    {
        static int s_variable;
        public:
            class FirstWithin
            {
                static int s_variable;
                public:
                    int value();
            
            int value();
            private:
                class SecondWithin
                {
                    static int s_variable;
                    public:
                        int value();
                }
    ```
If the class `Surround` should be able to access `FirstWithin` and `SecondWithin`'s private members, these latter two classes must declare `Surround` to be their friend. The function `Surround::value` can thereupon access the private members of its nested classes. For example (note the friend declarations in the two nested classes):

```cpp
class Surround
{
    static int s_variable;
    public:
        class FirstWithin
        {
            friend class Surround;
            static int s_variable;
            public:
                int value();
            int value();
        private:
            class SecondWithin
            {
                friend class Surround;
                static int s_variable;
                public:
                    int value();
            }
        }
    int value();

    inline int Surround::FirstWithin::value()
    {
        FirstWithin::s_variable = SecondWithin::s_variable;
        return (s_variable);
    }
};
```

Friend declarations may be provided beyond the definition of the entity that is to be considered a friend. So a class can be declared a friend beyond its definition. In that situation in-class code may already use the fact that it is going to be declared a friend by the upcoming class.

Note that members named identically in outer and inner classes (e.g., `s_variable`) may be accessed using the proper scope resolution expressions, as illustrated below:

```cpp
class Surround
{
    static int s_variable;
    public:
        class FirstWithin
        {
            friend class Surround;
            static int s_variable; // identically named
            public:
                int value();
        }
    int value();
```
17.3. ACCESSING PRIVATE MEMBERS IN NESTED CLASSES

private:
    class SecondWithin
    {
        friend class Surround;
        static int s_variable; // identically named
        public:
        int value();
    }
    static void classMember();
};
inline int Surround::value()
{
    // scope resolution expression
    FirstWithin::s_variable = SecondWithin::s_variable;
    return s_variable;
}
inline int Surround::FirstWithin::value()
{
    Surround::s_variable = 4; // scope resolution expressions
    Surround::classMember();
    return s_variable;
}
inline int Surround::SecondWithin::value()
{
    Surround::s_variable = 40; // scope resolution expression
    return s_variable;
}

Nested classes aren't automatically each other's friends. Here friend declarations must be applied to grant one nested classes access to another one's private members. To grant FirstWithin access to SecondWithin's private members a friend declaration in SecondWithin is required. But to grant SecondWithin access to FirstWithin's private members the class FirstWithin cannot simply use friend class SecondWithin, as SecondWithin's definition is as yet unknown.

Now a forward declaration of SecondWithin is required. This forward declaration must be provided by the class Surround, rather than by the class FirstWithin. It makes no sense to specify a forward declaration like 'class SecondWithin;' in the class FirstWithin itself, as this would refer to an external (global) class SecondWithin. SecondWithin's forward declaration can also not be specified inside FirstWithin as 'class Surround::SecondWithin;'. This attempt would generate the following error message:

'Surround' does not have a nested type named 'SecondWithin'

Instead of providing a forward declaration for SecondWithin inside the nested classes the class SecondWithin must be declared by the class Surround, before the class FirstWithin has been defined. This way SecondWithin's friend declaration is accepted inside FirstWithin. Here is an example in which all classes have full access to all private members of all involved classes:

class Surround
{
    // class SecondWithin; not required: friend declarations (see
    // below) double as forward declarations
    static int s_variable;
}
public:
    class FirstWithin
    {
        friend class Surround;
        friend class SecondWithin;
        static int s_variable;
        public:
            int value();
    };
    int value(); // implementation given above
private:
    class SecondWithin
    {
        friend class Surround;
        friend class FirstWithin;
        static int s_variable;
        public:
            int value();
    };
inline int Surround::FirstWithin::value()
{
    Surround::s_variable = SecondWithin::s_variable;
    return s_variable;
}
inline int Surround::SecondWithin::value()
{
    Surround::s_variable = FirstWithin::s_variable;
    return s_variable;
}

17.4 Nesting enumerations

Enumerations may also be nested in classes. Nesting enumerations is a good way to show the close connection between the enumeration and its class. Nested enumerations have the same controlled visibility as other class members. They may be defined in the private, protected or public sections of classes and are inherited by derived classes. In the class ios we’ve seen values like ios::beg and ios::cur. In the current Gnu C++ implementation these values are defined as values of the seek_dir enumeration:

class ios: public _ios_fields
{
    public:
        enum seek_dir
        {
            beg,
            cur,
            end
        };
};
As an illustration assume that a class `DataStructure` represents a data structure that may be traversed in a forward or backward direction. Such a class can define an enumeration `Traversal` having the values `FORWARD` and `BACKWARD`. Furthermore, a member function `setTraversal` can be defined requiring a `Traversal` type of argument. The class can be defined as follows:

```cpp
class DataStructure
{
    public:
        enum Traversal
        {
            FORWARD,
            BACKWARD
        }
        setTraversal(Traversal mode);  // ... do something
    private:
        Traversal d_mode;
};
```

Within the class `DataStructure` the values of the `Traversal` enumeration can be used directly. For example:

```cpp
void DataStructure::setTraversal(Traversal mode)
{
    d_mode = mode;
    switch (d_mode)
    {
        FORWARD:
            // ... do something
            break;
        BACKWARD:
            // ... do something else
            break;
    }
}
```

Outside of the class `DataStructure` the name of the enumeration type is not used to refer to the values of the enumeration. Here the classname is sufficient. Only if a variable of the enumeration type is required the name of the enumeration type is needed, as illustrated by the following piece of code:

```cpp
void fun()
{
    DataStructure::Traversal // enum typename required
    localMode = DataStructure::FORWARD; // enum typename not required

    DataStructure ds;  // enum typename not required
    ds.setTraversal(DataStructure::BACKWARD);
}
```

In the above example the constant `DataStructure::FORWARD` was used to specify a value of an enum defined in the class `DataStructure`. Instead of `DataStructure::FORWARD` the construction...
ds.FORWARD is also accepted. In my opinion this syntactic liberty is ugly: FORWARD is a symbolic value that is defined at the class level; it’s not a member of ds, which is suggested by the use of the member selector operator.

Only if DataStructure defines a nested class Nested, in turn defining the enumeration Traversal, the two class scopes are required. In that case the latter example should have been coded as follows:

```cpp
void fun()
{
    DataStructure::Nested::Traversal
        localMode = DataStructure::Nested::FORWARD;

    DataStructure ds;

    ds.setTraversal(DataStructure::Nested::BACKWARD);
}
```

Here the construction `DataStructure::Nested::Traversal localMode = ds.Nested::FORWARD` could also be used. I would avoid it.

### 17.4.1 Empty enumerations

Enum types usually define symbolic values. However, this is not required. In section 14.6.1 the `std::bad_cast` type was introduced. A bad_cast is thrown by the `dynamic_cast<>` operator when a reference to a base class object cannot be cast to a derived class reference. The bad_cast could be caught as type, irrespective of any value it might represent.

Types may be defined without any associated values. An empty enum can be defined which is an enum not defining any values. The empty enum’s type name may thereupon be used as a legitimate type in, e.g. a catch clause.

The example shows how an empty enum is defined (often, but not necessarily within a class) and how it may be thrown (and caught) as exceptions:

```cpp
#include <iostream>

enum EmptyEnum
{
};

int main()
try
{
    throw EmptyEnum();
}
catch (EmptyEnum)
{  
    std::cout << "Caught empty enum\n";
}
17.5 Revisiting virtual constructors

In section 14.12 the notion of virtual constructors was introduced. In that section a class Base was defined as an abstract base class. A class Clonable was defined to manage Base class pointers in containers like vectors.

As the class Base is a minute class, hardly requiring any implementation, it can very well be defined as a nested class in Clonable. This emphasizes the close relationship between Clonable and Base. Nesting Base under Clonable changes

    class Derived: public Base

into:

    class Derived: public Clonable::Base

Apart from defining Base as a nested class and deriving from Clonable::Base rather than from Base (and providing Base members with the proper Clonable:: prefix to complete their fully qualified names), no further modifications are required. Here are the modified parts of the program shown earlier (cf. section 14.12), now using Base nested under Clonable:

```cpp
// Clonable and nested Base, including their inline members:
class Clonable
{
    public:
        class Base;
    private:
        Base *d_bp;
    public:
        class Base
        {
            public:
                virtual ~Base();
                Base *clone() const;
            private:
                virtual Base *newCopy() const = 0;
            }
        Clonable();
        explicit Clonable(Base *base);
        ~Clonable();
        Clonable(Clonable const &other);
        Clonable(Clonable &&tmp);
        Clonable &operator=(Clonable const &other);
        Clonable &operator=(Clonable &&tmp);
    Base &base() const;
};
inline Clonable::Base *Clonable::Base::clone() const
{
    return newCopy();
}
inline Clonable::Base &Clonable::base() const
{
```
return *d_bp;
}

// Derived and its inline member:
class Derived1: public Clonable::Base
{
  public:
    Derived1();
  private:
    virtual Clonable::Base *newCopy() const;
  
  inline Clonable::Base *Derived1::newCopy() const
  {
    return new Derived1(*this);
  }

  // Members not implemented inline:
  Clonable::Base::~Base()
  {}
Chapter 18

The Standard Template Library

The Standard Template Library (STL) is a general purpose library consisting of containers, generic algorithms, iterators, function objects, allocators, adaptors and data structures. The data structures used by the algorithms are abstract in the sense that the algorithms can be used with (practically) any data type.

The algorithms can process these abstract data types because they are template based. This chapter does not cover template construction (see chapter 21 for that). Rather, it focuses on the use of the algorithms.

Several elements also used by the standard template library have already been discussed in the C++ Annotations. In chapter 12 abstract containers were discussed, and in section 11.10 function objects were introduced. Also, iterators were mentioned at several places in this document.

The main components of the STL are covered in this and the next chapter. Iterators, adaptors, smart pointers, multi threading and other features of the STL are discussed in coming sections. Generic algorithms are covered in the next chapter (19).

 Allocators take care of the memory allocation within the STL. The default allocator class suffices for most applications, and is not further discussed in the C++ Annotations.

All elements of the STL are defined in the standard namespace. Therefore, a using namespace std or a comparable directive is required unless it is preferred to specify the required namespace explicitly. In header files the std namespace should explicitly be used (cf. section 7.12.1).

In this chapter the empty angle bracket notation is frequently used. In code a typename must be supplied between the angle brackets. E.g., plus<> is used in the C++ Annotations, but in code plus<string> may be encountered.

18.1 Predefined function objects

Before using the predefined function objects presented in this section the <functional> header file must be included.

Function objects play important roles in generic algorithms. For example, there exists a generic algorithm sort expecting two iterators defining the range of objects that should be sorted, as well as a function object calling the appropriate comparison operator for two objects. Let's take a quick look at this situation. Assume strings are stored in a vector, and we want to sort the vector in
descending order. In that case, sorting the vector \texttt{stringVec} is as simple as:

\begin{verbatim}
    sort(stringVec.begin(), stringVec.end(), greater<string>());
\end{verbatim}

The last argument is recognized as a constructor: it is an instantiation of the \texttt{greater<>} class template, applied to \texttt{strings}. This object is called as a function object by the \texttt{sort} generic algorithm. The generic algorithm calls the function object's \texttt{operator()} member to compare two string objects. The function object's \texttt{operator()} will, in turn, call \texttt{operator>} of the \texttt{string} data type. Eventually, when \texttt{sort} returns, the first element of the vector will contain the string having the greatest \texttt{string} value of all.

The function object's \texttt{operator()} itself is not visible at this point. Don't confuse the parentheses in the '\texttt{greater<string>()}' argument with calling \texttt{operator()}. When \texttt{operator()} is actually used inside \texttt{sort}, it receives two arguments: two strings to compare for 'greaterness'. Since \texttt{greater<string>::operator()} is defined inline, the call itself is not actually present in the above \texttt{sort} call. Instead \texttt{sort} calls \texttt{string::operator>} through \texttt{greater<string>::operator()}.

Now that we know that a constructor is passed as argument to (many) generic algorithms, we can design our own function objects. Assume we want to sort our vector case-insensitively. How do we proceed? First we note that the default \texttt{string::operator<} (for an incremental sort) is not appropriate, as it does case sensitive comparisons. So, we provide our own \texttt{CaseInsensitive} class, which compares two strings case insensitively. Using the POSIX function \texttt{strcasicmp}, the following program performs the trick. It case-insensitively sorts its command-line arguments in ascending alphabetic order:

\begin{verbatim}
#include <iostream>
#include <string>
#include <cstring>
#include <algorithm>
using namespace std;

class CaseInsensitive
{
  public:
    bool operator()(string const &left, string const &right) const
    {
      return strcasecmp(left.c_str(), right.c_str()) < 0;
    }
};

int main(int argc, char **argv)
{
  sort(argv, argv + argc, CaseInsensitive());
  for (int idx = 0; idx < argc; ++idx)
    cout << argv[idx] << " ";
  cout << '\n';
}
\end{verbatim}

The default constructor of the \texttt{class CaseInsensitive} is used to provide \texttt{sort} with its final argument. So the only member function that must be defined is \texttt{CaseInsensitive::operator()}. Since we know it's called with \texttt{string} arguments, we define it to expect two \texttt{string} arguments, which are used when calling \texttt{strcasicmp}. Furthermore, \texttt{operator()} function is defined inline, so that it does not produce overhead when called by the \texttt{sort} function. The \texttt{sort} function calls the function object with various combinations of \texttt{strings}. If the compiler grants our inline requests, it will in fact call \texttt{strcasicmp}, skipping two extra function calls.
The comparison function object is often a pre-defined function object. Pre-defined function object classes are available for many commonly used operations. In the following sections the available pre-defined function objects are presented, together with some examples showing their use. Near the end of the section about function objects, function adaptors are introduced.

Pre-defined function objects are used predominantly with generic algorithms. Pre-defined function objects exist for arithmetic, relational, and logical operations. In section 24.3 pre-defined function objects are developed performing bitwise operations.

18.1.1 Arithmetic function objects

The arithmetic function objects support the standard arithmetic operations: addition, subtraction, multiplication, division, modulo and negation. These function objects invoke the corresponding operators of the data types for which they are instantiated. For example, for addition the function object plus<Type> is available. If we replace Type by size_t then the addition operator for size_t values is used, if we replace Type by string, the addition operator for strings is used. For example:

```cpp
#include <iostream>
#include <string>
#include <functional>
using namespace std;

int main(int argc, char **argv)
{
    plus<size_t> uAdd; // function object to add size_ts
    cout << "3 + 5 = " << uAdd(3, 5) << '\n';
    plus<string> sAdd; // function object to add strings
    cout << "argv[0] + argv[1] = " << sAdd(argv[0], argv[1]) << '\n';
}

/*
Output when called as: a.out going
3 + 5 = 8
argv[0] + argv[1] = a.outgoing
*/
```

Why is this useful? Note that the function object can be used with all kinds of data types (not only with the pre-defined datatypes) supporting the operator called by the function object.

Suppose we want to perform an operation on a left hand side operand which is always the same variable and a right hand side argument for which, in turn, all elements of an array should be used. E.g., we want to compute the sum of all elements in an array; or we want to concatenate all the strings in a text-array. In situations like these function objects come in handy.

As stated, function objects are heavily used in the context of the generic algorithms, so let’s take a quick look ahead at yet another one.

The generic algorithm accumulate visits all elements specified by an iterator-range, and performs a requested binary operation on a common element and each of the elements in the range, returning the accumulated result after visiting all elements specified by the iterator range. It’s easy to use this algorithm. The next program accumulates all command line arguments and prints the final string:
#include <iostream>
#include <string>
#include <functional>
#include <numeric>
using namespace std;

int main(int argc, char **argv)
{
    string result =
        accumulate(argv, argv + argc, string(), plus<string>());
    cout << "All concatenated arguments: " << result << '\n';
}

The first two arguments define the (iterator) range of elements to visit, the third argument is string. This anonymous string object provides an initial value. We could also have used

string("All concatenated arguments: ")

in which case the cout statement could simply have been cout << result << '\n'. The string-addition operation is used, called from plus<string>. The final concatenated string is returned.

Now we define a class Time, overloading operator+. Again, we can apply the predefined function object plus, now tailored to our newly defined datatype, to add times:

#include <iostream>
#include <string>
#include <vector>
#include <functional>
#include <numeric>
using namespace std;

class Time
{
    friend ostream &operator<<(ostream &str, Time const &time);
    size_t d_days;
    size_t d_hours;
    size_t d_minutes;
    size_t d_seconds;
    public:
        Time(size_t hours, size_t minutes, size_t seconds);
        Time &operator+=(Time const &rValue);
};

Time operator+(Time const &lValue, Time const &rValue)
{
    Time ret(lValue);
    ret += rValue;
    return ret;
}

Time::Time(size_t hours, size_t minutes, size_t seconds)
:
    d_days(0),
    d_hours(hours),
    d_minutes(minutes),

18.1. PREDEFINED FUNCTION OBJECTS

```
d_seconds(seconds)
{
}
Time &Time::operator+=(Time const &rValue)
{
    d_seconds += rValue.d_seconds;
    d_minutes += rValue.d_minutes + d_seconds / 60;
    d_hours += rValue.d_hours + d_minutes / 60;
    d_days += rValue.d_days + d_hours / 24;
    d_seconds %= 60;
    d_minutes %= 60;
    d_hours %= 24;
    return *this;
}
ostream &operator<<(ostream &str, Time const &time)
{
    return cout << time.d_days << " days, " << time.d_hours << " hours, " <<
    time.d_minutes << " minutes and " << time.d_seconds << " seconds.";
}
int main(int argc, char **argv)
{
    vector<Time> tvector;
    tvector.push_back(Time(1, 10, 20));
    tvector.push_back(Time(10, 30, 40));
    tvector.push_back(Time(20, 50, 0));
    tvector.push_back(Time(30, 20, 30));
    cout << accumulate
    (tvector.begin(), tvector.end(), Time(0, 0, 0), plus<Time>()) <<
    '\n';
    // Displays: 2 days, 14 hours, 51 minutes and 30 seconds.
```

The design of the above program is fairly straightforward. Time defines a constructor, it defines an insertion operator and it defines its own operator+, adding two time objects. In main four Time objects are stored in a vector<Time> object. Then, accumulate is used to compute the accumulated time. It returns a Time object, which is inserted into cout.

While the first example did show the use of a named function object, the last two examples showed the use of anonymous objects that were passed to the (accumulate) function.

The STL supports the following set of arithmetic function objects. The function call operator (operator()) of these function objects calls the matching arithmetic operator for the objects that are passed to the function call operator, returning that arithmetic operator’s return value. The arithmetic operator that is actually called is mentioned below:

- plus<>:: calls the binary operator+;
- minus<>:: calls the binary operator-;
- multiplies<>:: calls the binary operator*;
• divides<>: calls operator/;
• modulus<>: calls operator%;
• negate<>: calls the unary operator -. This arithmetic function object is a unary function object as it expects one argument.

In the next example the transform generic algorithm is used to toggle the signs of all elements of an array. Transform expects two iterators, defining the range of objects to be transformed; an iterator defining the begin of the destination range (which may be the same iterator as the first argument); and a function object defining a unary operation for the indicated data type.

```cpp
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>
using namespace std;

int main(int argc, char**argv)
{
    int iArr[] = { 1, -2, 3, -4, 5, -6 };
    transform(iArr, iArr + 6, iArr, negate<int>().
    for (int idx = 0; idx < 6; ++idx)
        cout << iArr[idx] << ", ";
    cout << \\
} // Displays: -1, 2, -3, 4, -5, 6,
```

18.1.2 Relational function objects

The relational operators are called by the relational function objects. All standard relational operators are supported: ==, !=, >, >=, < and <=.

The STL supports the following set of relational function objects. The function call operator (operator()) of these function objects calls the matching relational operator for the objects that are passed to the function call operator, returning that relational operator's return value. The relational operator that is actually called is mentioned below:

• equal_to<>: calls operator==;
• not_equal_to<>: calls operator!=;
• greater<>: calls operator>;
• greater_equal<>: calls operator>=;
• less<>: this object's operator() member calls operator<;
• less_equal<>: calls operator<=.

An example using the relational function objects in combination with sort is:

```cpp
#include <iostream>
```
18.1. PREDEFINED FUNCTION OBJECTS

#include <string>
#include <functional>
#include <algorithm>

using namespace std;

int main(int argc, char **argv)
{
    sort(argv, argv + argc, greater_equal<string>());
    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << '
';
    sort(argv, argv + argc, less<string>());
    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << '
';
}

The example illustrates how strings may be sorted alphabetically and reversed alphabetically. By passing greater_equal<string> the strings are sorted in decreasing order (the first word will be the 'greatest'), by passing less<string> the strings are sorted in increasing order (the first word will be the 'smallest').

Note that argv contains char * values, and that the relational function object expects a string. The promotion from char const * to string is silently performed.

18.1.3 Logical function objects

The logical operators are called by the logical function objects. The standard logical operators are supported: and, or, and not.

The STL supports the following set of logical function objects. The function call operator (operator()) of these function objects calls the matching logical operator for the objects that are passed to the function call operator, returning that logical operator’s return value. The logical operator that is actually called is mentioned below:

- logical_and<>: calls operator&&;
- logical_or<>: calls operator||;
- logical_not<>: calls operator!.

An example using operator! is provided in the following trivial program, using transform to transform the logical values stored in an array:

#include <iostream>
#include <string>
#include <functional>
#include <algorithm>
using namespace std;

#include <functional>
#include <algorithm>
using namespace std;

int main(int argc, char **argv)
{
    sort(argv, argv + argc, greater_equal<string>());
    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << '
';
    sort(argv, argv + argc, less<string>());
    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << '
';
}
int main(int argc, char **argv)
{
    bool bArr[] = {true, true, true, false, false, false};
    size_t const bArrSize = sizeof(bArr) / sizeof(bool);

    for (size_t idx = 0; idx < bArrSize; ++idx)
        cout << bArr[idx] << " ";
    cout << '\n';

    transform(bArr, bArr + bArrSize, bArr, logical_not<bool>());

    for (size_t idx = 0; idx < bArrSize; ++idx)
        cout << bArr[idx] << " ";
    cout << '\n';
}

/*
 * Displays:
 *     1 1 1 0 0 0
 *     0 0 0 1 1 1
 */

18.1.4 Function adaptors

Function adaptors modify the working of existing function objects. The STL offers three kinds of function adaptors: binders, negators and member function wrappers. Binders and negators are described in the next two subsections; member function adaptors are covered in section 19.2 of the next chapter, which is a more natural point of coverage than the current chapter.

18.1.4.1 Binders

Binders are function adaptors converting binary function objects to unary function objects. They do so by binding one parameter of a binary function object to a constant value. For example, the first parameter of the minus<int> function object may be bound to 100, meaning that the resulting value is always equal to 100 minus the value of the function object’s second argument.

Either the first or the second parameter may be bound to a specific value. To bind the constant value to the function object’s first parameter the function adaptor bind1st is used. To bind the constant value to the function object’s second parameter the function adaptor bind2nd is used. As an example, assume we want to count all elements of a vector of string objects that occur later in the alphabetical ordering than some reference string.

The count_if generic algorithm is the algorithm of choice for solving these kinds of problems. It expects the usual iterator range and a function object. However, instead of providing it with a function object it is provided with the bind2nd adaptor which in turn is initialized with a relational function object (greater<string>) and a reference string against which all strings in the iterator range are compared. Here is the required bind2nd specification:

    bind2nd(greater<string>(), referenceString)

Here is what this binder does:
To begin with, an adaptor (and so a binder) is a function object, so it defines `operator()` In this case the binder function object is a unary function object.

The binder’s `operator()` receives each of the strings referred to by the iterator range in turn.

Next it passes these strings and the binder’s second argument (the reference object) to the (binary) `operator()` defined by the function object that is passed as the binder’s first argument.

It returns the return value of that latter function object.

Although binders are defined as templates, it is illustrative to have a look at their implementations, assuming they were ordinary classes. Here is such a pseudo-implementation of the `bind2nd` function adaptor:

```cpp
class bind2nd
{
    FunctionObject d_object;
    Operand const &d_operand;
public:
    bind2nd(FunctionObject const &object, Operand const &operand);
    ReturnType operator()(Operand const &lvalue);
};
inline bind2nd::bind2nd(FunctionObject const &object,
    Operand const &operand)
{
    d_object(object),
    d_operand(operand)
}
inline ReturnType bind2nd::operator()(Operand const &lvalue)
{
    return d_object(lvalue, d_operand);
}
```

The binder’s `operator()` merely calls the function object’s `operator()`, providing it with two arguments. It uses its parameter as the (latter) `operator()`’s first argument and it uses `d_operand` as `operator()`’s second argument. The adaptor’s members are typically very small so they are usually implemented inline.

The above application of the `bind2nd` adaptor has another important characteristic. Its return type is identical to the return type of the function object that it receives as its first argument, which is `bool`. Functions returning `bool` values are also called *predicate functions*. In the above application the `bind2nd` adaptor therefore becomes a predicate function itself.

The `count_if` generic algorithm visits all the elements in an iterator range, returning the number of times the predicate specified as its final argument returns `true`. Each of the elements of the iterator range is passed to this predicate, which is therefore a unary predicate. Through the binder the binary function object `greater<>` is adapted to a unary function object, that now compares each of the elements referred to by the iterator range to the reference string. Eventually, the `count_if` function is called like this:

```cpp
count_if(stringVector.begin(), stringVector.end(),
    bind2nd(greater<string>(), referenceString));
```
18.1.4.2 Negators

Negators are function adaptors converting the values returned by predicate function. Since there are unary and binary predicate functions, two negator function adaptors were predefined: \texttt{not1} is the negator to use with unary predicates, \texttt{not2} is the negator to with binary function objects.

Example: to count the number of persons in a \texttt{vector<string>} vector ordered alphabetically before (i.e., \texttt{not} exceeding) a certain reference text one of the following alternatives could be used:

- a binary predicate that directly offers the required comparison:

\[
\text{count\_if(stringVector.begin(), stringVector.end(),}
\text{bind2nd(less\_equal<string>\(), referenceText))}
\]

- \texttt{not2} in combination with the \texttt{greater<>} predicate:

\[
\text{count\_if(stringVector.begin(), stringVector.end(),}
\text{bind2nd(not2(greater<string>\(), referenceText))}
\]

Here \texttt{not2} is used as it negates the truth value of a binary \texttt{operator()} member function.

- \texttt{not1} in combination with the \texttt{bind2nd} predicate:

\[
\text{count\_if(stringVector.begin(), stringVector.end(),}
\text{not1(bind2nd(greater<string>\(), referenceText)))}
\]

Here \texttt{not1} is used as it negates the truth value of a unary \texttt{operator()} member function.

The use of negators is illustrated by the following program:

```cpp
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
using namespace std;

int main(int argc, char **argv)
{
    int iArr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    cout << count_if(iArr, iArr + 10, bind2nd(less_equal<int>\(), 6))
    << \
    \ ';
    cout << count_if(iArr, iArr + 10, bind2nd(not2(greater<int>\(), 6))
    << \
    \ ';
    cout << count_if(iArr, iArr + 10, not1(bind2nd(greater<int>\(), 6)))
    << \
    \n';
    // Displays: 6 6 6
}
```

One may wonder which of these alternative approaches is the faster. Using the first approach, in which a directly available function object was used, two actions must be performed for each iteration by \texttt{count\_if}:

- The binder's \texttt{operator()} is called;
The operation $\leq$ is performed.

When the compiler uses inline as requested, only the second step is actually performed.

Using the second approach, using the `not2` negator to negate the truth value of the complementary logical function object, three actions must be performed for each iteration by `count_if`:

- The binder's `operator()` is called;
- The negator's `operator()` is called;
- The operation $>$ is performed.

When the compiler uses inline as requested, only the third step is actually performed.

Using the third approach, using `not1` negator to negate the truth value of the binder, three actions must be performed for each iteration by `count_if`:

- The negator's `operator()` is called;
- The binder's `operator()` is called;
- The operation $>$ is performed.

When the compiler uses inline as requested, only the third step is actually performed.

With a commonly used optimization flag like `-O2` the compiler tries to grant inline requests. However, if the compiler ignores the inline requests the first variant will be faster.

### 18.2 Iterators

In addition to the conceptual iterator types presented in this section the STL defines several adaptors allowing objects to be passed as iterators. These adaptors are presented in the upcoming sections. Before those adaptors can be used the `<iterator>` header file must be included.

Iterators are objects acting like pointers. Iterators have the following general characteristics:

- Two iterators may be compared for (in)equality using the `==` and `!=` operators. The `ordering` operators (e.g., `>`, `<`) can usually not be used.
- Given an iterator `iter`, `*iter` represents the object the iterator points to (alternatively, `iter->` can be used to reach the members of the object the iterator points to).
- `++iter` or `iter++` advances the iterator to the next element. The notion of advancing an iterator to the next element is consequently applied: several containers support `reversed_iterator` types, in which the `++iter` operation actually reaches a previous element in a sequence.
- Pointer arithmetic may be used with iterators of containers storing their elements consecutively in memory like `vector` and `deque`. For such containers `iter + 2` points to the second element beyond the one to which `iter` points. See also section 18.2.1, covering `std::distance`. 
• Merely defining an iterator is comparable to having a 0-pointer. Example:

```cpp
#include <vector>
#include <iostream>
using namespace std;

int main()
{
    vector<int>::iterator vi;
    cout << &*vi;  // prints 0
}
```

STL containers usually define members offering iterators (i.e., they define their own type iterator). These members are commonly called begin and end and (for reversed iterators (type reverse_iterator)) rbegin and rend.

Standard practice requires iterator ranges to be left inclusive. The notation \([\text{left}, \text{right})\) indicates that left is an iterator pointing to the first element, while right is an iterator pointing just beyond the last element. The iterator range is empty when left \(==\) right.

The following example shows how all elements of a vector of strings can be inserted into cout using its iterator ranges \([\text{begin()}, \text{end()}),\) and \([\text{rbegin()}, \text{rend()}).\) Note that the for-loops for both ranges are identical. Furthermore it nicely illustrates how the auto keyword can be used to define the type of the loop control variable instead of using a much more verbose variable definition like vector<string>::iterator (see also section 3.3.5):

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main(int argc, char **argv)
{
    vector<string> args(argv, argv + argc);
    for (auto iter = args.begin(); iter != args.end(); ++iter)
        cout << *iter << " ";
    cout << '\n';
    for (auto iter = args.rbegin(); iter != args.rend(); ++iter)
        cout << *iter << " ";
    cout << '\n';
}
```

Furthermore, the STL defines const_iterator types that must be used when visiting a series of elements in a constant container. Whereas the elements of the vector in the previous example could have been altered, the elements of the vector in the next example are immutable, and const_iterators are required:

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;
```
int main(int argc, char **argv)
{
    vector<string> const args(argv, argv + argc);

    for
    {
        vector<string>::const_iterator iter = args.begin();
        iter != args.end();
        ++iter
    }
    cout << *iter << " ";
    cout << '\n';

    for
    {
        vector<string>::const_reverse_iterator iter = args.rbegin();
        iter != args.rend();
        ++iter
    }
    cout << *iter << " ";
    cout << '\n';

    return 0;
}

The examples also illustrates that plain pointers can be used as iterators. The initialization
vector<string> args(argv, argv + argc) provides the args vector with a pair of pointer-based iterators: argv points to the first element to initialize args with, argv + argc points just beyond the last element to be used, ++argv reaches the next command line argument. This is a
general pointer characteristic, which is why they too can be used in situations where iterators
are expected.

The STL defines five types of iterators. These iterator types are expected by generic algorithms, and
in order to create a particular type of iterator yourself it is important to know their characteristics.
In general, iterators (see also section 22.14) must define:

- operator==, testing two iterators for equality,
- operator!=, testing two iterators for inequality,
- operator++, incrementing the iterator, as prefix operator,
- operator*, to access the element the iterator refers to,

The following types of iterators are used when describing generic algorithms in chapter 19:

- **InputIterators**:

  InputIterators are used to read from a container. The dereference operator is guaranteed
to work as rvalue in expressions. Instead of an InputIterator it is also possible
to use (see below) Forward-, Bidirectional- or RandomAccessIterators. Notations like
InputIterator1 and InputIterator2 may be used as well. In these cases, numbers
are used to indicate which iterators ‘belong together’. E.g., the generic algorithm
inner_product has the following prototype:
Type inner_product(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, Type init);

InputIterator1 first1 and InputIterator1 last1 define a pair of input iterators on one range, while InputIterator2 first2 defines the beginning of another range. Analogous notations may be used with other iterator types.

- **OutputIterators**:

  OutputIterators can be used to write to a container. The dereference operator is guaranteed to work as an *lvalue* in expressions, but not necessarily as *rvalue*. Instead of an OutputIterator it is also possible to use (see below) Forward-, Bidirectional- or RandomAccessIterators.

- **ForwardIterators**:

  ForwardIterators combine InputIterators and OutputIterators. They can be used to traverse containers in one direction, for reading and/or writing. Instead of a ForwardIterator it is also possible to use (see below) Bidirectional- or RandomAccessIterators.

- **BidirectionalIterators**:

  BidirectionalIterators can be used to traverse containers in both directions, for reading and writing. Instead of a BidirectionalIterator it is also possible to use (see below) a RandomAccessIterator.

- **RandomAccessIterators**:

  RandomAccessIterators provide random access to container elements. An algorithm like *sort* requires a RandomAccessIterator, and can therefore *not* be used to sort the elements of lists or maps, which only provide BidirectionalIterators.

The example given with the RandomAccessIterator illustrates how to relate iterators and generic algorithms: look for the iterator that’s required by the (generic) algorithm, and then see whether the datastructure supports the required type of iterator. If not, the algorithm cannot be used with the particular datastructure.

### 18.2.1 `std::distance`

Earlier, in section 18.2 it was stated that iterators support pointer arithmetic for containers storing their elements consecutively in memory. This is not completely true: to determine the number of elements between the elements to which two iterators refer the iterator must support the subtraction operator.

Using pointer arithmetic to compute the number of elements between two iterators in, e.g., a `std::list` or `std::unordered_map` is not possible, as these containers do not store their elements consecutively in memory.

The function `std::distance` fills in that little gap: `std::distance` expects two InputIterators and returns the number of elements between them.

Before using `distance` the `<iterator>` header file must be included.

If the iterator specified as first argument exceeds the iterator specified as its second argument then the number of elements is non-positive, otherwise it is non-negative. If the number of elements cannot be determined (e.g., the iterators do not refer to elements in the same container), then `distance`’s return value is undefined.
Example:

```cpp
#include <iostream>
#include <unordered_map>

using namespace std;

int main()
{
    unordered_map<int, int> myMap = {{1, 2}, {3, 5}, {-8, 12}};
    cout << distance(++myMap.begin(), myMap.end()) << \n'; // shows: 2
}
```

### 18.2.2 Insert iterators

Generic algorithms often require a target container into which the results of the algorithm are deposited. For example, the `copy` generic algorithm has three parameters. The first two define the range of visited elements, the third defines the first position where the results of the copy operation should be stored.

With the `copy` algorithm the number of elements to copy is usually available beforehand, since that number can usually be provided by pointer arithmetic. However, situations exist where pointer arithmetic cannot be used. Analogously, the number of resulting elements sometimes differs from the number of elements in the initial range. The generic algorithm `unique_copy` is a case in point. Here the number of elements that are copied to the destination container is normally not known beforehand.

In situations like these an `inserter` adaptor function can often be used to create elements in the destination container. There are three types of inserter adaptors:

- **back_inserter**: calls the container's `push_back` member to add new elements at the end of the container. E.g., to copy all elements of `source` in reversed order to the back of `destination`, using the `copy` generic algorithm:

  ```cpp
  copy(source.rbegin(), source.rend(), back_inserter(destination));
  ```

- **front_inserter** calls the container's `push_front` member, adding new elements at the beginning of the container. E.g., to copy all elements of `source` to the front of the destination container (thereby also reversing the order of the elements):

  ```cpp
  copy(source.begin(), source.end(), front_inserter(destination));
  ```

- **inserter** calls the container's `insert` member adding new elements starting at a specified starting point. E.g., to copy all elements of `source` to the destination container, starting at the beginning of `destination`, shifting up existing elements to beyond the newly inserted elements:

  ```cpp
  copy(source.begin(), source.end(), inserter(destination, destination.begin()));
  ```

The inserter adaptors require the existence of two `typedefs:`
• typedef Data value_type, where Data is the data type stored in the class offering push_back, push_front or insert members (Example: typedef std::string value_type);

• typedef value_type const &const_reference

Concentrating on back_inserter, this iterator expects the name of a container supporting a member push_back. The inserter’s operator() member calls the container’s push_back member. Objects of any class supporting a push_back member can be passed as arguments to back_inserter provided the class adds

typedef DataType const &const_reference;

to its interface (where DataType const & is the type of the parameter of the class’s member push_back). Example:

#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

class Insertable
{
    public:
        typedef int value_type;
        typedef int const &const_reference;

        void push_back(int const &)
        {
        }
};

int main()
{
    int arr[] = {1};
    Insertable insertable;

    copy(arr, arr + 1, back_inserter(insertable));
}

18.2.3 Iterators for ‘istream’ objects

The istream_iterator<Type> can be used to define a set of iterators for istream objects. The general form of the istream_iterator iterator is:

    istream_iterator<Type> identifier(istream &in)

Here, Type is the type of the data elements read from the istream stream. It is used as the ‘begin’ iterator in an interator range. Type may be any type for which operator>> is defined in combination with istream objects.

The default constructor is used as the end-iterator and corresponds to the end-of-stream. For example,
The `stream` object that was specified when defining the begin-iterator is not mentioned with the default constructor.

Using `back_inserter` and `istream_iterator` adaptors, all strings from a stream can easily be stored in a container. Example (using anonymous `istream_iterator` adaptors):

```c++
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<string> vs;
    copy(istream_iterator<string>(cin), istream_iterator<string>(),
         back_inserter(vs));

    for
    {
        vector<string>::const_iterator begin = vs.begin(), end = vs.end();
        begin != end; ++begin
    }
    cout << *begin << ' ';
    cout << '
';
}
```

### 18.2.3.1 Iterators for ‘istreambuf’ objects

Input iterators are also available for `streambuf` objects.

To read from `streambuf` objects supporting input operations `istreambuf_iterators` can be used, supporting the operations that are also available for `istream_iterator`. Different from the latter iterator type `istreambuf_iterators` support three constructors:

- **`istreambuf_iterator<Type>`:**
  
  The end iterator of an iterator range is created using the default `istreambuf_iterator` constructor. It represents the end-of-stream condition when extracting values of type `Type` from the `streambuf`.

- **`istreambuf_iterator>Type>(streambuf *)**:
  
  A pointer to a `streambuf` may be used when defining an `istreambuf_iterator`. It represents the begin iterator of an iterator range.

- **`istreambuf_iterator<Type>(istream)`**:
  
  An `istream` may be also used when defining an `istreambuf_iterator`. It accesses the `istream`'s `streambuf` and it also represents the begin iterator of an iterator range.
In section 18.2.4.1 an example is given using both istreambuf_iterators and ostreambuf_iterators.

18.2.4 Iterators for ‘ostream’ objects

An ostream_iterator<Type> adaptor can be used to pass an ostream to algorithms expecting an OutputIterator. Two constructors are available for defining ostream_iterators:

    ostream_iterator<Type> identifier(ostream &outStream);
    ostream_iterator<Type> identifier(ostream &outStream, char const *delim);

Type is the type of the data elements that should be inserted into an ostream. It may be any type for which operator<< is defined in combinations with ostream objects. The latter constructor can be used to separate the individual Type data elements by delimiter strings. The former constructor does not use any delimiters.

The example shows how istream_iterators and an ostream_iterator may be used to copy information of a file to another file. A subtlety here is that you probably want to use in.unsetf(ios::skipws). It is used to clear the ios::skipws flag. As a consequence white space characters are simply returned by the operator, and the file is copied character by character. Here is the program:

    #include <iostream>
    #include <algorithm>
    #include <iterator>
    using namespace std;

    int main()
    {
        cin.unsetf(ios::skipws);
        copy(istream_iterator<char>(cin), istream_iterator<char>(),
             ostream_iterator<char>(cout));
    }

18.2.4.1 Iterators for ‘ostreambuf’ objects

Output iterators are also available for streambuf objects.

To write to streambuf objects supporting output operations ostreambuf_iterators can be used, supporting the operations that are also available for ostream_iterator. Ostreambuf_iterators support two constructors:

- ostreambuf_iterator<Type>(streambuf *):
  
  A pointer to a streambuf may be used when defining an ostreambuf_iterator. It can be used as an OutputIterator.

- ostreambuf_iterator<Type>(ostream):
  
  An ostream may be also used when defining an ostreambuf_iterator. It accesses the ostream’s streambuf and it can also be used as an OutputIterator.
The next example illustrates the use of both istreambuf_iterators and ostreambuf_iterators when copying a stream in yet another way. Since the stream's streambufs are directly accessed the streams and stream flags are bypassed. Consequently there is no need to clear ios::skipws as in the previous section, while the next program's efficiency probably also exceeds the efficiency of the program shown in the previous section.

```cpp
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    istreambuf_iterator<char> in(cin.rdbuf());
    istreambuf_iterator<char> eof;
    ostreambuf_iterator<char> out(cout.rdbuf());

    copy(in, eof, out);
    return 0;
}
```

18.3 The class 'unique_ptr'

Before using the `unique_ptr` class presented in this section the `<memory>` header file must be included.

When pointers are used to access dynamically allocated memory strict bookkeeping is required to prevent memory leaks. When a pointer variable referring to dynamically allocated memory goes out of scope, the dynamically allocated memory becomes inaccessible and the program suffers from a memory leak.

To prevent such memory leaks strict bookkeeping is required: the programmer has to make sure that the dynamically allocated memory is returned to the common pool just before the pointer variable goes out of scope.

When a pointer variable points to a dynamically allocated single value or object, bookkeeping requirements are greatly simplified when the pointer variable is defined as a `std::unique_ptr` object.

`Unique_ptrs` are objects masquerading as pointers. Since they are objects, their destructors are called when they go out of scope. Their destructors automatically delete the dynamically allocated memory.

`Unique_ptrs` have some special characteristics:

- when assigning a `unique_ptr` to another move semantics is used. If move semantics is not available compilation fails. On the other hand, if compilation succeeds then the used containers or generic algorithms support the use of `unique_ptrs`. Here is an example:

```cpp
std::unique_ptr<int> up1(new int);
std::unique_ptr<int> up2(up1);  // compilation error
```
The second definition fails to compile as unique_ptr’s copy constructor is private (the same holds true for the assignment operator). But the unique_ptr class does offer facilities to initialize and assign from rvalue references:

```cpp
class unique_ptr // interface partially shown
{
    public:
        unique_ptr(unique_ptr &&other); // rvalues bind here
    private:
        unique_ptr(const unique_ptr &other);
};
```

In the next example move semantics is used and so it compiles correctly:

```cpp
unique_ptr<int> cp(unique_ptr<int>(new int));
```

- a unique_ptr object should only point to memory that was made available dynamically, as only dynamically allocated memory can be deleted.

- multiple unique_ptr objects should not be allowed to point to the same block of dynamically allocated memory. The unique_ptr’s interface was designed to prevent this from happening. Once a unique_ptr object goes out of scope, it deletes the memory it points to, immediately changing any other object also pointing to the allocated memory into a wild pointer.

- When a class Derived is derived from Base, then a newly allocated Derived class object can be assigned to a unique_ptr<Base>, without having to define a virtual destructor for Base. The Base * pointer that is returned by the unique_ptr object can simply be cast statically to Derived, as shown in the following example:

```cpp
class Base
{ ... };
class Derived: public Base
{
    // assume Derived has a member void process()
};
int main()
{
    shared_ptr<Base> bp(new Derived);
    static_cast<Derived *>(bp)->process(); // OK!
}
```

The class unique_ptr offers several member functions to access the pointer itself or to have a unique_ptr point to another block of memory. These member functions (and unique_ptr constructors) are introduced in the next few sections.

A unique_ptr (as well as a shared_ptr, see section 18.4) can be used as a safe alternative to the now deprecated auto_ptr. Unique_ptr also augments auto_ptr as it can be used with containers and (generic) algorithms as it adds customizable destructors. Arrays can also be handled by unique_ptrs.

### 18.3.1 Defining ‘unique_ptr’ objects

There are three ways to define unique_ptr objects. Each definition contains the usual <type> specifier between angle brackets:
18.3. THE CLASS 'UNIQUE_PTR'

- The default constructor simply creates a unique_ptr object that does not point to a particular block of memory. Its pointer is initialized to 0 (zero):

  ```cpp
  unique_ptr<type> identifier;
  ```

  This form is discussed in section 18.3.2.

- The move constructor initializes an unique_ptr object. Following the use of the move constructor its unique_ptr argument no longer points to the dynamically allocated memory and its pointer data member is turned into a zero-pointer:

  ```cpp
  unique_ptr<type> identifier(another unique_ptr for type);
  ```

  This form is discussed in section 18.3.3.

- The form that is used most often initializes a unique_ptr object to the block of dynamically allocated memory that is passed to the object's constructor. Optionally deleter can be provided. A (free) function (or function object) receiving the unique_ptr's pointer as its argument can be passed as deleter. It is supposed to return the dynamically allocated memory to the common pool (doing nothing if the pointer equals zero):

  ```cpp
  unique_ptr<type> identifier (new-expression [, deleter]);
  ```

  This form is discussed in section 18.3.4.

18.3.2 Creating a plain 'unique_ptr'

Unique_ptr's default constructor defines a unique_ptr not pointing to a particular block of memory:

  ```cpp
  unique_ptr<type> identifier;
  ```

The pointer controlled by the unique_ptr object is initialized to 0 (zero). Although the unique_ptr object itself is not the pointer, its value can be compared to 0. Example:

  ```cpp
  unique_ptr<int> ip;
  if (!ip)
    cout << "0-pointer with a unique_ptr object\n";
  ```

Alternatively, the member get can be used (cf. section 18.3.5).

18.3.3 Moving another 'unique_ptr'

A unique_ptr may be initialized using an rvalue reference to a unique_ptr object for the same type:

  ```cpp
  unique_ptr<type> identifier(other unique_ptr object);
  ```
The move constructor is used, e.g., in the following example:

```c++
void mover(unique_ptr<string> & & param)
{
    unique_ptr<string> tmp(move(param));
}
```

Analogously, the assignment operator can be used. A `unique_ptr` object may be assigned to a temporary `unique_ptr` object of the same type (again move-semantics is used). For example:

```c++
#include <iostream>
#include <memory>
#include <string>

using namespace std;

int main()
{
    unique_ptr<string> hello1(new string("Hello world"));
    unique_ptr<string> hello2(move(hello1));
    unique_ptr<string> hello3;
    hello3 = move(hello2);
    cout << "\n" << // would have segfaulted
         "\n" << // same
     *hello3 << '\n';
}

// Displays: Hello world
```

The example illustrates that:

- `hello1` is initialized by a pointer to a dynamically allocated `string` (see the next section).
- The `unique_ptr` `hello2` grabs the pointer controlled by `hello1` using a move constructor. This effectively changes `hello1` into a 0-pointer.
- Then `hello3` is defined as a default `unique_ptr<string>`. But then it grabs its value using move-assignment from `hello2` (which, as a consequence, is changed into a 0-pointer as well)

If `hello1` or `hello2` had been inserted into `cout` a segmentation fault would have resulted. The reason for this should now be clear: it is caused by dereferencing 0 pointers. In the end, only `hello3` actually points to the originally allocated `string`.

### 18.3.4 Pointing to a newly allocated object

A `unique_ptr` is most often initialized using a pointer to dynamically allocated memory. The generic form is:

```c++
unique_ptr<type [, deleter_type]> identifier(new-expression
     [, deleter = deleter_type()]);
```
18.3. THE CLASS 'UNIQUE_PTR'

The second (template) argument (deleter(_type)) is optional and may refer to a free function or function object handling the destruction of the allocated memory. A deleter is used, e.g., in situations where a double pointer is allocated and the destruction must visit each nested pointer to destroy the allocated memory (see below for an illustration).

Here is an example initializing a unique_ptr pointing to a string object:

```cpp
unique_ptr<string> strPtr(new string("Hello world"));
```

The argument that is passed to the constructor is the pointer returned by operator new. Note that type does not mention the pointer. The type that is used in the unique_ptr construction is the same as the type that is used in new expressions.

Here is an example showing how an explicitly defined deleter may be used to delete a dynamically allocated array of pointers to strings:

```cpp
#include <iostream>
#include <string>
#include <memory>
using namespace std;

struct Deleter
{
  size_t d_size;
  Deleter(size_t size = 0)
  :
    d_size(size)
  {}
  void operator()(string **ptr) const
  {
    for (size_t idx = 0; idx < d_size; ++idx)
      delete ptr[idx];
    delete[] ptr;
  }
};
int main()
{
  unique_ptr<string *, Deleter> sp2(new string *[10], Deleter(10));
  Deleter &obj = sp2.get_deleter();
}
```

A unique_ptr can be used to reach the member functions that are available for objects allocated by the new expression. These members can be reached as if the unique_ptr was a plain pointer to the dynamically allocated object. For example, in the following program the text 'C++' is inserted behind the word 'hello':

```cpp
#include <iostream>
#include <memory>
#include <cstring>
using namespace std;

int main()
{

}
unique_ptr<string> sp(new string("Hello world"));

cout << *sp << '\n';
sp->insert(strlen("Hello "), "C++ ");
cout << *sp << '\n';
}
/*
 * Displays:
 ** Hello world
 ** Hello C++ world
 */

18.3.5 Operators and members

The class unique_ptr offers the following operators:

- unique_ptr<Type> &operator=(unique_ptr<Type> &&tmp):
  
  This operator transfers the memory pointed to by the rvalue unique_ptr object to the lvalue unique_ptr object using move semantics. So, the rvalue object loses the memory it pointed at and turns into a 0-pointer. An existing unique_ptr may be assigned to another unique_ptr by converting it to an rvalue reference first using std::move. Example:

  ```cpp
  unique_ptr<int> ip1(new int);
  unique_ptr<int> ip2;
  ip2 = std::move(ip1);
  ```

- operator bool() const:
  
  This operator returns false if the unique_ptr does not point to memory (i.e., its get member, see below, returns 0). Otherwise, true is returned.

- Type &operator*():
  
  This operator returns a reference to the information accessible via a unique_ptr object. It acts like a normal pointer dereference operator.

- Type *operator->():
  
  This operator returns a pointer to the information accessible via a unique_ptr object. This operator allows you to select members of an object accessible via a unique_ptr object. Example:

  ```cpp
  unique_ptr<string> sp(new string("hello");
cout << sp->c_str();
  ```

The class unique_ptr supports the following member functions:

- Type *get():
  
  A pointer to the information controlled by the unique_ptr object is returned. It acts like operator->. The returned pointer can be inspected. If it is zero the unique_ptr object does not point to any memory.
18.3. THE CLASS ‘UNIQUE_PTR’

- **Deleter &unique_ptr<Type>::get_deleter():**
  A reference to the deleter object used by the unique_ptr is returned.

- **Type *release():**
  A pointer to the information accessible via a unique_ptr object is returned. At the same time the object itself becomes a 0-pointer (i.e., its pointer data member is turned into a 0-pointer). This member can be used to transfer the information accessible via a unique_ptr object to a plain Type pointer. After calling this member the proper destruction of the dynamically allocated memory is the responsibility of the programmer.

- **void reset(Type *):**
  The dynamically allocated memory controlled by the unique_ptr object is returned to the common pool; the object thereupon controls the memory to which the argument that is passed to the function points. It can also be called without argument, turning the object into a 0-pointer. This member function can be used to assign a new block of dynamically allocated memory to a unique_ptr object.

- **void swap(unique_ptr<Type> &):**
  Two identically typed unique_ptrs are swapped.

18.3.6 Using ‘unique_ptr’ objects for arrays

When a unique_ptr is used to store arrays the dereferencing operator makes little sense but with arrays unique_ptr objects benefit from index operators. The distinction between a single object unique_ptr and a unique_ptr referring to a dynamically allocated array of objects is realized through a template specialization.

With dynamically allocated arrays the following syntax is available:

- the index ([ ]) notation is used to specify that the smart pointer controls a dynamically allocated array. Example:
  ```cpp
  unique_ptr<int[]> intArr(new int[3]);
  ```

- the index operator can be used to access the array's elements. Example:
  ```cpp
  intArr[2] = intArr[0];
  ```

In these cases the smart pointer's destructors call `delete[]` rather than `delete`.

18.3.7 The legacy class 'auto_ptr' (deprecated)

The (now deprecated by the C++11 standard) smart pointer class `std::auto_ptr<Type>` has traditionally been offered by C++. This class does not support move semantics, but when an auto_ptr object is assigned to another, the right-hand object loses its information.

The class unique_ptr does not have auto_ptr's drawbacks and consequently using auto_ptr is now deprecated. Car_ptrs suffer from the following drawbacks:

- they do not support move semantics;
• they should not be used to point to arrays;
• they cannot be used as data types of abstract containers.

Because of its drawbacks and available replacements the auto_ptr class is no longer covered by
the C++ Annotations. Existing software should be modified to use smart pointers (unique_ptrs
or shared_ptrs) and new software should, where applicable, directly be implemented in terms of
these new smart pointer types.

18.4 The class ’shared_ptr’

In addition to unique_ptr the C++11 standard offers std::shared_ptr<Type> which is a refer-
ence counting smart pointer.

Before using shared_ptrs the <memory> header file must be included.

The shared pointer automatically destroys its contents once its reference count has decayed to zero.
As with unique_ptr, when defining a shared_ptr<Base> to store a newly allocated Derived class
object, the returned Base * may be cast to a Derived * using a static_cast: polymorphism isn’t
required, and when resetting the shared_ptr or when the shared_ptr goes out of scope, no slicing
occurs, and Derived’s destructor is called (cf. section 18.3).

Shared_ptrs support copy and move constructors as well as standard and move overloaded assign-
ment operators.

Like unique_ptrs, shared_ptrs may refer to dynamically allocated arrays.

18.4.1 Defining ‘shared_ptr’ objects

There are four ways to define shared_ptr objects. Each definition contains the usual <type>
specifier between angle brackets:

• The default constructor simply creates a shared_ptr object that does not point to a particular
  block of memory. Its pointer is initialized to 0 (zero):

  shared_ptr<type> identifier;

  This form is discussed in section 18.4.2.

• The copy constructor initializes a shared_ptr so that both objects share the memory pointed
  at by the existing object. The copy constructor also increments the shared_ptr’s reference
  count. Example:

  shared_ptr<string> org(new string("hi there"));
  shared_ptr<string> copy(org); // reference count now 2

• The move constructor initializes a shared_ptr with the pointer and reference count of a
temporary shared_ptr. The temporary shared_ptr is changed into a 0-pointer. An ex-
sting shared_ptr may have its data moved to a newly defined shared_ptr (turning the
existing shared_ptr into a 0-pointer as well). In the next example a temporary, anonymous
shared_ptr object is constructed, which is then used to construct grabber. Since grabber’s
18.4. THE CLASS ‘SHARED_PTR’

constructor receives an anonymous temporary object, the compiler uses shared_ptr’s move constructor:

```cpp
shared_ptr<string> grabber(shared_ptr<string>(new string("hi there")));
```

- The form that is used most often initializes a shared_ptr object to the block of dynamically allocated memory that is passed to the object’s constructor. Optionally deleter can be provided. A (free) function (or function object) receiving the shared_ptr’s pointer as its argument can be passed as deleter. It is supposed to return the dynamically allocated memory to the common pool (doing nothing if the pointer equals zero).

```cpp
shared_ptr<type> identifier (new-expression [, deleter]);
```

This form is discussed in section 18.4.3.

18.4.2 Creating a plain ‘shared_ptr’

Shared_ptr’s default constructor defines a shared_ptr not pointing to a particular block of memory:

```cpp
shared_ptr<type> identifier;
```

The pointer controlled by the shared_ptr object is initialized to 0 (zero). Although the shared_ptr object itself is not the pointer, its value can be compared to 0. Example:

```cpp
shared_ptr<int> ip;
if (!ip) 
  cout << "0-pointer with a shared_ptr object\n";
```

Alternatively, the member get can be used (cf. section 18.4.4).

18.4.3 Pointing to a newly allocated object

Most often a shared_ptr is initialized by a dynamically allocated block of memory. The generic form is:

```cpp
shared_ptr<type> identifier(new-expression [, deleter]);
```

The second argument (deleter) is optional and refers to a function object or free function handling the destruction of the allocated memory. A deleter is used, e.g., in situations where a double pointer is allocated and the destruction must visit each nested pointer to destroy the allocated memory (see below for an illustration). It is used in situations comparable to those encountered with unique_ptr (cf. section 18.3.4).

Here is an example initializing a shared_ptr pointing to a string object:

```cpp
shared_ptr<string> strPtr(new string("Hello world"));
```
The argument that is passed to the constructor is the pointer returned by `operator new`. Note that `type` does not mention the pointer. The type that is used in the `shared_ptr` construction is the same as the type that is used in `new` expressions.

The next example illustrates that two `shared_ptrs` indeed share their information. After modifying the information controlled by one of the objects the information controlled by the other object is modified as well:

```cpp
#include <iostream>
#include <memory>
#include <cstring>
using namespace std;

int main()
{
    shared_ptr<string> sp(new string("Hello world"));
    shared_ptr<string> sp2(sp);

    sp->insert(strlen("Hello "), "C++ ");
    cout << *sp << \n << *sp2 << \n;
}
/*
Displays:
Hello C++ world
Hello C++ world
*/
```

### 18.4.4 Operators and members

The class `shared_ptr` offers the following operators:

- **`shared_ptr &operator=(shared_ptr<Type> const &other)`**:
  
  Copy assignment: the reference count of the operator's left hand side operand is reduced. If the reference count decays to zero the dynamically allocated memory controlled by the left hand side operand is deleted. Then it shares the information with the operator's right hand side operand, incrementing the information's reference count.

- **`shared_ptr &operator=(shared_ptr<Type> &&tmp)`**:
  
  Move assignment: the reference count of the operator's left hand side operand is reduced. If the reference count decays to zero the dynamically allocated memory controlled by the left hand side operand is deleted. Then it grabs the information controlled by the operator's right hand side operand which is turned into a 0-pointer.

- **`operator bool()` const**:
  
  If the `shared_ptr` actually points to memory `true` is returned, otherwise, `false` is returned.

- **`Type &operator*()`**:
  
  A reference to the information stored in the `shared_ptr` object is returned. It acts like a normal pointer.
18.4. THE CLASS 'SHARED_PTR'

- Type *operator->():
  A pointer to the information controlled by the shared_ptr object is returned. Example:
  ```cpp
  shared_ptr<string> sp(new string("hello"));
  cout << sp->c_str() << '\n';
  ```

The following member function member functions are supported:

- Type *get():
  A pointer to the information controlled by the shared_ptr object is returned. It acts like operator->. The returned pointer can be inspected. If it is zero the shared_ptr object does not point to any memory.

- Deleter &get_deleter():
  A reference to the shared_ptr’s deleter (function or function object) is returned.

- void reset(Type *):
  The reference count of the information controlled by the shared_ptr object is reduced and if it decays to zero the memory it points to is deleted. Thereafter the object’s information will refer to the argument that is passed to the function, setting its shared count to 1. It can also be called without argument, turning the object into a 0-pointer. This member function can be used to assign a new block of dynamically allocated memory to a shared_ptr object.

- void shared_ptr<Type>::swap(shared_ptr<Type> &&):
  Two identically typed shared_ptrs are swapped.

- bool unique() const:
  If the current object is the only object referring to the memory controlled by the object true is returned otherwise (including the situation where the object is a 0-pointer) false is returned.

- size_t use_count() const:
  The number of objects sharing the memory controlled by the object is returned.

18.4.5 Casting shared pointers

Be cautious when using standard C++ style casts in combination with shared_ptr objects. Consider the following two classes:

```cpp
struct Base
{
};
struct Derived: public Base
{};
```

As with unique_ptr, when defining a shared_ptr<Base> to store a newly allocated Derived class object, the returned Base * may be cast to a Derived * using a static_cast: polymorphism isn’t required, and when resetting the shared_ptr or when the shared_ptr goes out of scope, no slicing occurs, and Derived’s destructor is called (cf. section 18.3).
Of course, a `shared_ptr<Derived>` can easily be defined. Since a `Derived` object is also a `Base` object, a pointer to `Derived` can be considered a pointer to `Base` without using casts, but a `static_cast` could be used for force the interpretation of a `Derived *` to a `Base *`:

```cpp
Derived d;
static_cast<Base *>(&d);
```

However, a plain `static_cast` cannot be used when initializing a shared pointer to a `Base` using the `get` member of a shared pointer to a `Derived` object. The following code snippet eventually results in an attempt to delete the dynamically allocated `Base` object twice:

```cpp
shared_ptr<Derived> sd(new Derived);
shared_ptr<Base> sb(static_cast<Base *>(sd.get()));
```

Since `sd` and `sb` point at the same object, the `~Base` destructor will be called for the same object when `sb` goes out of scope and when `sd` goes out of scope, resulting in premature termination of the program due to a `double free` error.

These errors can be prevented using casts that were specifically designed for being used with shared pointers. These casts use specialized constructors that create a shared pointer pointing to memory but shares ownership (i.e., a reference count) with an existing shared pointer. These special casts are:

- `std::static_pointer_cast<Base>(std::shared_ptr<Derived> ptr)`:
  A shared pointer to a `Base` class object is returned. The returned shared pointer refers to the base class portion of the `Derived` class to which the `shared_ptr<Derived>` `ptr` refers. Example:
  ```cpp
  shared_ptr<Derived> dp(new Derived());
  shared_ptr<Base> bp = static_pointer_cast<Base>(dp);
  ```

- `std::const_pointer_cast<Class>(std::shared_ptr<Class const> ptr)`:
  A shared pointer to a `Class` class object is returned. The returned shared pointer refers to a non-const `Class` object whereas the `ptr` argument refers to a `Class` const object. Example:
  ```cpp
  shared_ptr<Derived const> cp(new Derived());
  shared_ptr<Derived> ncp = const_pointer_cast<Derived>(cp);
  ```

- `std::dynamic_pointer_cast<Derived>(std::shared_ptr<Base> ptr)`:
  A shared pointer to a `Derived` class object is returned. The `Base` class must have at least one virtual member function, and the class `Derived`, inheriting from `Base`, may have overridden `Base`'s virtual member(s). The returned shared pointer refers to a `Derived` class object if the dynamic cast from `Base *` to `Derived *` succeeded. If the dynamic cast did not succeed the shared pointer's `get` member returns 0. Example (assume `Derived` and `Derived2` were derived from `Base`):
  ```cpp
  shared_ptr<Base> bp(new Derived());
  cout << dynamic_pointer_cast<Derived>(bp).get() << ' ' <<
      dynamic_pointer_cast<Derived2>(bp).get() << '\n';
  ```
  The first `get` returns a non-0 pointer value, the second `get` returns 0.
18.4.6 Using ‘shared_ptr’ objects for arrays

Different from the unique_ptr class no specialization exists for the shared_ptr class to handle dynamically allocated arrays of objects.

But like unique_ptrs, with shared_ptrs referring to arrays the dereferencing operator makes little sense while in these circumstances shared_ptr objects would benefit from index operators.

It is not difficult to create a class shared_array offering such facilities. The class template shared_array, derived from shared_ptr merely should provide an appropriate deleter to make sure that the array and its elements are properly destroyed. In addition it should define the index operator and optionally could declare the derefencing operators using delete.

Here is an example showing how shared_array can be defined and used:

```cpp
struct X
{
    ~X()
    {
        cout << "destr\n"; // show the object’s destruction
    }
};
template<typename Type>
class shared_array: public shared_ptr<Type>
{
    struct Deleter // Deleter receives the pointer
    {
        void operator()(Type* ptr)
        {
            delete[] ptr;
        }
    }
    public:
    shared_array(Type *p) // other constructors
    { // not shown here
        shared_ptr<Type>(p, Deleter())
    }
    Type &operator[](size_t idx) // index operators
    {
        return shared_ptr<Type>::get()[idx];
    }
    Type const &operator[](size_t idx) const
    {
        return shared_ptr<Type>::get()[idx];
    }
    Type &operator*() = delete; // delete pointless members
    Type const &operator*() const = delete;
    Type *operator->() = delete;
    Type const *operator->() const = delete;
};
int main()
{
    shared_array<X> sp(new X[3]);
    sp[0] = sp[1];
}
```
Usually a `shared_ptr` is initialized at definition time with a pointer to a newly allocated object. Here is an example:

```cpp
std::shared_ptr<std::string> sptr(new std::string("hello world"));
```

In such statements two memory allocation calls are used: one for the allocation of the `std::string` and one used interally by `std::shared_ptr`'s constructor itself.

The two allocations can be combined into one single allocation (which is also slightly more efficient than explicitly calling `shared_ptr`'s constructor) using the `make_shared` template. The function template `std::make_shared` has the following prototype:

```cpp
template<typename Type, typename ...Args>
std::shared_ptr<Type> std::make_shared(Args ...args);
```

Before using `make_shared` the `<memory>` header file must be included.

This function template allocates an object of type `Type`, passing `args` to its constructor (using `perfect forwarding`, see section 22.5.2), and returns a `shared_ptr` initialized with the address of the newly allocated `Type` object.

Here is how the above `sptr` object can be initialized using `std::make_shared`. Notice the use of `auto` which frees us from having to specify `sptr`'s type explicitly:

```cpp
auto sptr(std::make_shared<std::string>("hello world"));
```

After this initialization `std::shared_ptr<std::string> sptr` has been defined and initialized. It could be used as follows:

```cpp
std::cout << *sptr << '\n';
```

### 18.6 Classes having pointer data members

Classes having pointer data members require special attention. In particular at construction time one must be careful to prevent wild pointers and/or memory leaks. Consider the following class defining two pointer data members:

```cpp
class Filter
{
    istream *d_in;
    ostream *d_out;
    public:
        Filter(char const *in, char const *out);
};
```

Assume that `Filter` objects filter information read from `*d_in` and write the filtered information to `*d_out`. Using pointers to streams allows us to have them point at any kind of stream like...
18.6. CLASSES HAVING POINTER DATA MEMBERS

istreams, ifstreams, ofstreams or istringstream. The shown constructor could be implemented like this:

```cpp
Filter::Filter(char const *in, char const *out)
    :
      d_in(new ifstream(in)),
      d_out(new ofstream(out))
{
  if (!*d_in || !*d_out)
    throw string("Input and/or output stream not available");
}
```

Of course, the construction could fail. `new` could throw an exception; the stream constructors could throw exceptions; or the streams could not be opened in which case an exception is thrown from the constructor's body. Using a function try block helps. Note that if `d_in`'s initialization throws, there's nothing to be worried about. The `Filter` object hasn't been constructed, its destructor is not be called and processing continues at the point where the thrown exception is caught. But `Filter`'s destructor is also not called when `d_out`'s initialization or the constructor's if statement throws: no object, and hence no destructor is called. This may result in memory leaks, as `delete` isn't called for `d_in` and/or `d_out`. To prevent this, `d_in` and `d_out` must first be initialized to 0 and only then the initialization can be performed:

```cpp
Filter::Filter(char const *in, char const *out)
    try
    :
      d_in(0),
      d_out(0)
{
  d_in = new ifstream(in);
  d_out = new ofstream(out);
  if (!*d_in || !*d_out)
    throw string("Input and/or output stream not available");
}
    catch (...)
    {
      delete d_out;
      delete d_in;
    }
```

This quickly gets complicated, though. If `Filter` harbors yet another data member of a class whose constructor needs two streams then that data cannot be constructed or it must itself be converted into a pointer:

```cpp
Filter::Filter(char const *in, char const *out)
    try
    :
      d_in(0),
      d_out(0)
      d_filterImp(*d_in, *d_out)  // won't work
    {
      ...
    }
    // instead:
```
Filter::Filter(char const *in, char const *out)  
try  
  :  
    d_in(0),  
    d_out(0),  
    d_filterImp(0)  
  {  
    d_in = new ifstream(in);  
    d_out = new ofstream(out);  
    d_filterImp = new FilterImp(*d_in, *d_out);  
    ...  
  }  
  catch (...)  
  {  
    delete d_filterImp;  
    delete d_out;  
    delete d_in;  
  }

Although the latter alternative works, it quickly gets hairy. In situations like these smart pointers should be used to prevent the hairiness. By defining the stream pointers as (smart pointer) objects they will, once constructed, properly be destroyed even if the rest of the constructor's code throws exceptions. Using a FilterImp and two unique_ptr data members Filter's setup and its constructor becomes:

class Filter  
{  
  std::unique_ptr<std::ifstream> d_in;  
  std::unique_ptr<std::ofstream> d_out;  
  FilterImp d_filterImp;  
  ...  
};

Filter::Filter(char const *in, char const *out)  
try  
  :  
    d_in(new ifstream(in)),  
    d_out(new ofstream(out)),  
    d_filterImp(*d_in, *d_out)  
  {  
    if (!*d_in || !*d_out)  
      throw string("Input and/or output stream not available");  
  }

We're back at the original implementation but this time without having to worry about wild pointers and memory leaks. If one of the member initializers throws the destructors of previously constructed data members (which are now objects) are always called.

As a rule of thumb: when classes need to define pointer data members they should define those pointer data members as smart pointers if there's any chance that their constructors throw exceptions.
18.7 Lambda expressions

The C++11 standard adds lambda expressions to the language. As we’ll see in chapter 19 generic algorithms often accept arguments that can either be function objects or plain functions. Examples are the sort (cf. section 19.1.58) and find_if (cf. section 19.1.16) generic algorithms. As a rule of thumb: when a called function must remember its state a function object is appropriate, otherwise a plain function can be used.

Frequently the function or function object is not readily available, and it must be defined in or near the location where it is used. This is commonly realized by defining a class or function in the anonymous namespace (say: class or function A), passing an A to the code needing A. If that code is itself a member function of the class B, then A’s implementation might benefit from having access to the members of class B.

Often this scheme results in a significant amount of code (defining the class), or it results in complex code (to make available software elements that aren’t automatically accessible to A’s code). It may also result in code that is irrelevant at the current level of specification. Nested classes don’t solve these problems either. Moreover, nested classes can’t be used in templates.

Lambda expressions solve these problems. A lambda expression defines an anonymous function object, also called a closure object. When a lambda expression is evaluated it results in a temporary object (the closure object). The type of a closure object is called its closure type.

Lambda expressions may be used inside blocks, classes or namespaces (i.e., pretty much anywhere you like). Their implied closure type is defined in the smallest block, class or namespace scope which contains the lambda expression. The closure object’s visibility starts at its point of definition and ends where its closure type ends.

The closure type defines a (const) public inline function call operator. Here is an example of a lambda expression:

```cpp
[](int x, int y) { return x * y; }
```

The function call operator of the closure object created by this lambda expression expects two int arguments and returns their product. It is an inline const member of the closure type. To drop the const attribute, the lambda expression should specify mutable, as follows:

```cpp
[](int x, int y) mutable ...
```

The lambda-declarator may be omitted, if no parameters are defined. The parameters in a lambda declarator may not be given default arguments.

A closure object as defined by the above lambda expression could be used e.g., in combination with the accumulate (cf. section 19.1.1) generic algorithm to compute the product of a series of int values stored in a vector:

```cpp
cout << accumulate(vi.begin(), vi.end(), 1,
    [](int x, int y) { return x * y; });
```
The above lambda function uses the implicit return type `decltype(x * y)`. An implicit return type can be used if the lambda expression does not contain a `return` statement (i.e., a void lambda expression), if it contains a single `return` statement, or if it contains multiple `return` statements returning values of identical types (e.g., all `int` values).

If there are multiple `return` statements returning values of different types then the lambda expression's return type must specified be explicitly using a late-specified return type, (cf. section 3.3.5):

```cpp
[](int x, int y) -> int
{
    if (y < 0)
        return x / static_cast<double>(y);
    return z + x;
}
```

Variables that are visible at the location of a lambda expression can be accessed by the lambda expression. How these variables are accessed depends on the contents of the lambda-introducer (the area between the square brackets, called the the lambda-capture). The lambda-capture allows passing a local context to lambda expressions.

Visible global and static variables as well as local variables defined in the lambda expression's compound statement itself can directly be accessed and, if applicable, modified. Example:

```cpp
int global;

void fun()
{
    [](int) // [] may contain any specification
    {
        int localVariable = 0;
        localVariable = ++global;
    }
}
```

If the lambda expression is defined within a (non-static) class member function then an initial `&` or `=` character in the lambda-capture enables this, allowing the lambda expression access to all class members (data and functions). The class's data members can be modified.

If the lambda expression is defined inside a function then that function's local variables that are visible at the point of the lambda expression's definition can be accessed by the lambda expression. An initial `&` character in the lambda-capture accesses these local variables by reference. These variables can be modified from within the lambda expression.

An initial `=` character in the lambda-capture creates a local copy of the referred-to local variables. Furthermore, in this case the values of these local copies can only be changed by the lambda expression if the lambda expression is defined using the `mutable` keyword. E.g.,

```cpp
struct Class
{
    void fun()
    {
```
18.7. LAMBDA EXPRESSIONS

```cpp
int var = 0;
[=]() mutable {
    ++var; // modifies the local
} // copy, not fun's var
```

Fine-tuning is also possible. With an initial `=`, comma-separated `&var` specifications indicate that the mentioned local variables should be processed by reference, rather than as copies; with an initial `&`, comma separated `var` specifications indicate that local copies should be used of the mentioned local variables. Again, these copies have immutable values unless the lambda expression is provided with the `mutable` keyword.

Here is an example, using the `for_each` (cf. section 19.1.17) generic algorithm:

```cpp
void showSum(vector<int> const &vi)
{
    int total = 0;
    for_each(
        vi.begin(), vi.end(),
        [](int x)
        {
            total += x;
        });
    std::cout << total << '
';
}
```

The variable `int total` is passed to the lambda expression by reference and is directly accessed by the function. Its parameter list merely defines an `int x`, which is initialized in sequence by each of the values stored in `vi`. Once the generic algorithm has completed `showSum`'s variable `total` has received a value that is equal to the sum of all the vector's values. It has outlived the lambda expression and its value is displayed.

Although generic algorithms are extremely useful, they may not always be available for the task at hand. Moreover, an algorithm like `for_each` looks a bit unwieldy, now that the language offers range-based for-loops. So let's try this, instead of the above implementation:

```cpp
void showSum(vector<int> const &vi)
{
    int total = 0;
    for (auto el: vi)
        total += x;
    std::cout << total << '
';
}
```

But when `showSum` is now called, its `cout` statement consistently reports 0. What's happening here?

When a generic algorithm is given a lambda function, its implementation instantiates a reference to a function, and that function is thereupon called from within the generic algorithm. However,
in the above example the range-based for-loop’s nested statement merely represents the declaration of a lambda function: nothing is actually called, and hence total remains equal to 0. So, to make the above example work we not only must define the lambda expression, but we must also call the lambda function. One way of doing that is by giving the lambda function a name, and then call the lambda function by its given name:

```cpp
void showSum(vector<int> const &vi)
{
    int total = 0;
    for (auto el: vi)
    {
        auto lambda = [&](int x) {
            total += x;
        };

        lambda(el);
    }
    std::cout << total << '
';
}
```

In fact, there is no need to give the lambda function a name: the auto lambda declaration merely represents the lambda function, which could also directly be called, as an extension of its definition. The syntax may look a bit weird, but there’s nothing wrong with it, and it allows us to drop the compound statement, required in the last example, completely. Here goes:

```cpp
void showSum(vector<int> const &vi)
{
    int total = 0;
    for (auto el: vi) {
        total += x;
    })(el); // immediately append the argument list to the lambda function’s definition
    std::cout << total << '
';
}
```

Another fine-tuning uses this in the lambda-capture: it also allows the lambda-expression to access the surrounding class members. Example:

```cpp
class Data
{
    std::vector<std::string> d_names;
public:
    void show() const
    {
        int count = 0;
        std::for_each(d_names.begin(), d_names.end(),
        [this, &count](std::string const &name) {
            std::cout << ++count << ' ' <<
        });
    }
};
```
Lambda expressions may also be assigned to variables. An example of such an assignment (using \texttt{auto} to define the variable's type) is:

\[
\text{auto } \text{ sqr } = [](\text{int } x) \\
\{ \\
\quad \text{return } x * x; \\
\};
\]

The lifetime of such lambda expressions is equal to the lifetime of the variable receiving the lambda expression as its value. Named lambda functions nicely fit in the niche of local functions: when a function needs to perform some computations that are at a conceptually lower level than the function's task itself, then it's attractive to encapsulate these computations in a separate support function, and call the support function where needed. A support function can be defined in an anonymous namespace, but that quickly becomes awkward when the requiring function is a class member, and the support function needs access to the class's members as well. In that case a named lambda expression can be used: it can be defined within the requiring function, and may be given full access to the surrounding class. The name to which the lambda expression is assigned becomes the name of a function which can be called from the surrounding function. Here is an example, converting a numeric IP address to a dotted decimal string, which can also be accessed directly from an \texttt{Dotted} object (all implementations in-class to conserve space):

\[
\text{class } \text{ Dotted} \\
\{ \\
\quad \text{std::string } \text{d_dotted}; \\
\}
\]

\[
\text{public:} \\
\quad \text{std::string const } \&\text{dotted()} \text{ const} \\
\quad \{ \\
\quad \quad \text{return } \text{d_dotted}; \\
\quad \} \\
\quad \text{std::string const } \&\text{dotted(size_t } \text{ip}) \\
\quad \{ \\
\quad \quad \text{auto } \text{octet } = \\
\quad \quad \quad [](\text{size_t } \text{idx}, \text{size_t } \text{numeric}) \\
\quad \quad \quad \{ \\
\quad \quad \quad \quad \text{return } \text{to_string}(\text{numeric } >> \text{idx } \ast \text{8 } \& \text{0xff}); \\
\quad \quad \quad \}; \\
\quad \quad \text{d_dotted } = \\
\quad \quad \quad \text{octet(3, } \text{ip} \} + \text{'.'} + \text{octet(2, } \text{ip} \} + \text{'.'} + \\
\quad \quad \quad \text{octet(1, } \text{ip} \} + \text{'.'} + \text{octet(0, } \text{ip} \}; \\
\quad \quad \text{return } \text{d_dotted}; \\
\};
\]
Now that lambda expressions have been covered let’s see how they can be used to avoid spurious returns from `condition_variable’s wait` calls (cf. section 20.5.3). According to the C++11 standard, condition variables may spuriously return from `wait` calls. Therefore it is necessary to check that the data are actually available once `wait` continues.

The class `condition_variable` allows us to do that by offering `wait` members expecting a lock and a predicate. The predicate checks the data’s state, and returns `true` if the data’s state allows the data’s processing. Here is an alternative implementation of the `down` member shown in section 20.5.3, checking for the data’s actual availability:

```cpp
void down()
{
    unique_lock<mutex> lock(sem_mutex);
    condition.wait(lock,
        [&]() {
            return semaphore != 0
        });
    --semaphore;
}
```

The lambda expression ensures that `wait` only returns once `semaphore` has been incremented.

## 18.8 Regular Expressions

The C++11 standard adds handling of regular expression regular expressions to the language. Before using regular expressions as offered by the C++ standard the header file `<regex>` must be included.

Regular expressions were already available in C++ via its C heritage as C has always offered functions like `regcomp` and `regexec`. C++ itself did not provide facilities for handling regular expressions. Programmers had to use the basic `std::string` matching procedures (like `find_first_of`), thus avoiding regular expressions, or they had to use the existing C functions (which do not encapsulate (manipulating) the required data structures), or regular expressions where available through regular expression classes made available by third-party libraries (like the `Pattern` class of the Bobcat\(^1\)) library.

Regular expressions are extensively documented elsewhere (e.g., `regex(7)`, Friedl, J.E.F *Mastering Regular Expressions*\(^2\), O’Reilly). The reader is referred to these sources for a refresher on the topic of regular expressions. In essence, regular expressions define a small meta-language recognizing textual units (like ‘numbers’, ‘identifiers’, etc.). They are extensively used in the context of lexical scanners (cf. section 24.8.1) when defining the sequence of input characters associated with *tokens*. But they are also intensively used in other situations. Programs like `sed(1)` and `grep(1)` use regular expressions to find pieces of text in files having certain characteristics, and a program like `perl(1)` adds some ‘sugar’ to the regular expression language, simplifying the construction of regular expressions. However, though extremely useful, it is also well known that regular expressions tend to be very hard to read. Some even call the regular expression language a *write-only language*: while specifying a regular expression it’s often clear why it’s written in a particular way. But the opposite, understanding what a regular expression is supposed to represent if you lack the proper context,

---

\(^1\)http://bobcat.sourceforge.net

\(^2\)http://oreilly.com/catalog/
can be extremely difficult. That’s why, from the onset and as a rule of thumb, it is stressed that an appropriate comment should be provided, with each regular expression, as to what it is supposed to match.

In the upcoming sections first a short overview of the regular expression language is provided, which is then followed by the facilities C++ is currently offering for using regular expressions. These facilities mainly consist of classes helping you to specify regular expression, matching them to text, and determining which parts of the text (if any) match (parts of) the text being analyzed.

### 18.8.1 The regular expression mini language

Regular expressions are expressions consisting of elements resembling those of numeric expressions. Regular expressions consist of basic elements and operators, having various priorities and associations. Like numeric expressions, parentheses can be used to group elements together to form a units on which operators operate. For an extensive discussion the reader is referred to, e.g., section 15.10 of the ecma-international.org page, which describes the characteristics of the regular expressions used by default by C++’s regex classes.

C++’s default definition of regular expressions distinguishes the following atoms:

- \textbf{x}: the character ‘x’;
- \textbf{.}: any character except for the newline character;
- \textbf{[xyz]}: a character class; in this case, either an ‘x’, a ‘y’, or a ‘z’ matches the regular expression. See also the paragraph about character classes below;
- \textbf{[abj-oZ]}: a character class containing a range of characters; this regular expression matches an ‘a’, a ‘b’, any letter from ‘j’ through ‘o’, or a ‘Z’. See also the paragraph about character classes below;
- \textbf{[^A-Z]}: a negated character class: this regular expression matches any character but those in the class beyond ^. In this case, any character except for an uppercase letter. See also the paragraph about character classes below;
- \textbf{[:predef:]}, a predefined set of characters. See below for an overview. When used, it is interpreted as an element in a character class. It is therefore always embedded in a set of square brackets defining the character class (e.g., [[:alnum:]])
- \textbf{\X}: if X is ‘a’, ‘b’, ‘f’, ‘n’, ‘r’, ‘t’, or ‘v’, then the ANSI-C interpretation of ‘\x’. Otherwise, a literal ‘X’ (used to escape operators such as +);\n- \textbf{(r)}: the regular expression \textit{r}. It is used to override precedence (see below), but also to define \textit{r} as a marked sub-expression whose matching characters may directly be retrieved from, e.g., an std::smatch object (cf. section 18.8.3);\n- \textbf{(?r)}: the regular expression \textit{r}. It is used to override precedence (see below), but it is not regarded as a marked sub-expression;

In addition to these basic atoms, the following special atoms are available (which can also be used in character classes):

- \textbf{\s}: a white space character;\n
\footnote{http://ecma-international.org/ecma-262/5.1/#sec-15.10}
- \S: any character but a white space character;
- \d: a decimal digit character;
- \D: any character but a decimal digit character;
- \w: an alphanumeric character or an underscore (_) character;
- \W: any character but an alphanumeric character or an underscore (_) character.

Atoms may be concatenated. If \textit{r} and \textit{s} are atoms then the regular expression \textit{rs} matches a target text if the target text matches \textit{r} and \textit{s}, in that order (without any intermediate characters inside the target text). E.g., the regular expression \texttt{[ab][cd]} matches the target text \texttt{ac}, but not the target text \texttt{a:c}.

Atoms may be combined using operators. Operators bind to the preceding atom. If an operator should operate on multiple atoms the atoms must be surrounded by parentheses (see the last element in the previous itemization). To use an operator character as an atom it can be escaped. Eg., \* represent an operator, \* the atom character star. Note that character classes do not recognize escape sequences: \texttt{[\*]} represents a character class consisting of two characters: a backslash and a star.

The following operators are supported (\textit{r} and \textit{s} represent regular expression atoms):

- \textit{r*}: zero or more \textit{rs};
- \textit{r+}: one or more \textit{rs};
- \textit{r?}: zero or one \textit{rs} (that is, an optional \textit{r});
- \textit{r{m, n}}: where \texttt{1 <= m <= n}: matches \textquoteleft r\textquoteleft at least \textit{m}, but at most \textit{n} times;
- \textit{r{m}}: where \texttt{1 <= m}: matches \textquoteleft r\textquoteleft exactly \textit{m} times;
- \textit{r{m, n}}: where \texttt{1 <= m}: matches \textquoteleft r\textquoteleft exactly \textit{m} times;
- \textit{\^r}: \^ is a pseudo operator. This expression matches \textquoteleft r\textquoteleft, if appearing at the beginning of the target text. If the \^-character is not the first character of a regular expression it is interpreted as a literal \^-character;
- \textit{r$}: $ is a pseudo operator. This expression matches \textquoteleft r\textquoteleft, if appearing at the end of the target text. If the $-character is not the last character of a regular expression it is interpreted as a literal $-character;

When a regular expression contains marked sub-expressions and multipliers, and the marked sub-expressions are multiply matched, then the target’s final sub-string matching the marked sub-expression is reported as the text matching the marked sub-expression. E.g., when using \texttt{regex_search} (cf. section 18.8.4.2), marked sub-expression ((\texttt{(a|b)}+\texttt{\$?})) and target text \texttt{a a b}, then \texttt{a a b} is the fully matched text, while \texttt{b} is reported as the sub-string matching the first and second marked sub-expressions.
18.8. REGULAR EXPRESSIONS

18.8.1.1 Character classes

Inside a character class all regular expression operators lose their special meanings, except for the special atoms \s, \S, \d, \D, \w, and \W; the character range operator -; the end of character class operator ]; and, at the beginning of the character class, ^. Except in combination with the special atoms the escape character is interpreted as a literal backslash character (to define a character class containing a backslash and a d simply use \[d\]).

To add a closing bracket to a character class use [ ] immediately following the initial open-bracket, or start with [^] for a negated character class not containing the closing bracket. Minus characters are used to define character ranges (e.g., [a-d], defining [abcd]) (be advised that the actual range may depend on the locale being used). To add a literal minus character to a character class put it at the very beginning ([-, or [^] or at the very end (- ) ) of a character class.

Once a character class has started, all subsequent characters are added to the class’s set of characters, until the final closing bracket ( ) has been reached.

In addition to characters and ranges of characters, character classes may also contain predefined sets of character. They are:

[:alnum:] [:alpha:] [:blank:]
[:cntrl:] [:digit:] [:graph:]
[:lower:] [:print:] [:punct:]
[:space:] [:upper:] [:xdigit:]

These predefined sets designate sets of characters equivalent to the corresponding standard C isXXX function. For example, [:alnum:] defines all characters for which isalnum(3) returns true.

18.8.2 Defining regular expressions: std::regex

Before using the (w)regex class presented in this section the <regex> header file must be included.

The types std::regex and std::wregex define regular expression patterns. They define, respectively the types basic_regex<char> and basic_regex<wchar_t> types. Below, the class regex will be presented, but in the examples wregex could also have been used.

Regular expression facilities were, to a large extent, implemented using templates, using, e.g., the basic_string<char> type (which is equal to std::string). Likewise, generic types like OutputIterator (output iterator) and BidirConstIterator (bidirectional const iterator) are used with several functions. Such functions are function templates. Function templates determine the actual types from the arguments that are provided at call-time.

These are the steps that are commonly involved when using regular expressions:

• First, a regular expression is defined. This involves defining or modifying a regex object.

• Then the regular expression is provided with a target text, which may result in sections of the target text matching the regular expression.

• The sections of the target text matching (or not matching) the regular expression are retrieved to be processed elsewhere, or:

• The sections of the target text matching (or not matching) the regular expression are directly modified by existing regular expression facilities, after which the modified target text may be processed elsewhere.
The way `regex` objects handle regular expressions is defined by a `bit_or` combined set of `std::regex_constants` values, resulting in a `regex::flag_type` value. These `regex_constants` are:

- `std::regex_constants::awk`:
  `awk(1)’s (POSIX) regular expression grammar is used to specify regular expressions;`

- `std::regex_constants::basic`:
  `the basic POSIX regular expression grammar is used to specify regular expressions;`

- `std::regex_constants::collate`:
  `the character range operator (−) used in character classes defines a locale sensitive range (e.g., [a-k]);`

- `std::regex_constants::ECMAScript`:
  `the flag_type which is used by default by regex constructors. The regular expression uses the Modified ECMAScript regular expression grammar;`

- `std::regex_constants::egrep`:
  `egrep(1)’s (POSIX) regular expression grammar is used to specify regular expressions. This is the same grammar as used by regex_constants::extended, with the addition of the newline character (’\n’) as an alternative for the ’|’-operator;`

- `std::regex_constants::extended`:
  `the extended POSIX regular expression grammar is used to specify regular expressions;`

- `std::regex_constants::grep`:
  `grep(1)’s (POSIX) regular expression grammar is used to specify regular expressions. This is the same grammar as used by regex_constants::basic, with the addition of the newline character (’\n’) as an alternative for the ’|’-operator;`

- `std::regex_constants::icase`:
  `letter casing in the target string is ignored. E.g., the regular expression A matches a and A;`

- `std::regex_constants::nosub`:
  `When performing matches, all sub-expressions ((expr)) are treated as non-marked (?:expr);`

- `std::regex_constants::optimize`:
  `optimizes the speed of matching regular expressions, at the cost of slowing down the construction of the regular expression somewhat. If the same regular expression object is frequently used then this flag may substantially improve the speed of matching target texts;`
Constructors

Except for the default, move and copy constructors (which are all available) all constructors accept an additional argument of type `regex::flag_type`. If not specified the value `regex_constants::ECMAScript` is used.

- `regex()`:
  the default constructor defines a `regex` object not containing a regular expression;
- `explicit regex(char const *pattern)`: defines a `regex` object containing the regular expression found at `pattern`;
- `regex(char const *pattern, std::size_t count)`:
  defines a `regex` object containing the regular expression found at the first `count` characters of `pattern`;
- `explicit basic_regex(std::string const &pattern)`: defines a `regex` object containing the regular expression found at `pattern`. This constructor is defined as a member template, accepting a `basic_string`-type argument which may also use non-standard character traits and allocators;
- `regex(ForwardIterator first, ForwardIterator last)`: defines a `regex` object containing the regular expression found at the (forward) iterator range `[first, last)`. This constructor is defined as a member template, accepting any forward iterator type which can be used to define the regular expression’s pattern;
- `regex(std::initializer_list<Char> init)`: defines a `regex` object containing the regular expression from the characters in the initializer list `init`.

Here are some examples:

```cpp
std::regex re("\\w+"); // matches a sequence of alpha-numeric // and/or underscore characters

std::regex re({'\', 'w', '+'}); // idem

std::regex re(R"(\w+xxx")", 3); // idem
```

Member functions

- `regex &operator=(RHS)`:
  The copy and move assignment operators are available. Otherwise, RHS may be:
  - an NTBS (of type `const char *`);
  - a `std::string const &` (or any compatible `std::basic_string`);
  - a `std::initializer_list<char>`;
- `regex &assign(RHS)`:
  This member accepts the same arguments as `regex`’s constructors, including the (optional) `regex_constants` values;
• regex::flag_type flag() const:
  
  Returns the regex_constants flags that are active for the current regex object. E.g.,

  ```cpp
  int main()
  {
    regex::flag_type flags = re.flags();

    cout <<
      // displays: 32 1 0
      (re.flags() & regex_constants::basic) << ' ' <<
      (re.flags() & regex_constants::icase) << ' ' <<
      (re.flags() & regex_constants::awk) << ' ' << '
    ;
  }
  ```

• locale_type get_loc() const:

  Returns the locale that is associated with the current regex object;

• locale_type imbue(locale_type locale):

  Replaces the regex object’s current locale setting with locale, returning the replaced locale;

• unsigned mark_count() const:

  The number of marked sub-expressions in the regex object is returned. E.g.,

  ```cpp
  int main()
  {
    regex re("\w+\[[[:alpha:]]+\]");
    cout << re.mark_count() << '\n';  // displays: 2
  }
  ```

• void swap(regex &other) noexcept:

  Swaps the current regex object with other. Also available as a free function: void swap(regex &lhs, regex &rhs), swapping lhs and rhs.

### 18.8.3 Retrieving matches: std::match_results

Once a regex object is available, it can be used to match the regular expression against some target text. Several functions are available performing such matches, like `regex_match`, `regex_search`, and `regex_replace`, which are covered in the next section.

Usually the result of a regular expression matching process is made available in an object of the class `std::match_results`, covered in this section. Before using the `regex_match` class presented in this section the `<regex>` header file must be included.

Comparable to regex’s specializations, specializations of `match_results` exist. These specializations differ in the types of iterators or pointers that are used by `match_results`. They are:

• `cmatch`:

  defines `match_results<char const *>, using a char const * type of ‘iterator’;`

• `wcmatch`:

  defines `match_results<wchar_t const *>, using a wchar_t const * type of ‘iterator’;`
18.8. REGULAR EXPRESSIONS

- **smatch**:  
  defines `match_results<std::string::const_iterator>`, using a `std::string::const_iterator` type of iterator;

- **wsmatch**:  
  defines `match_results<std::wstring::const_iterator>`, using a `std::wstring::const_iterator` type of iterator.

**Constructors**

The class `match_results` only offers the standard default, copy, and move constructors. The default constructor actually defines one parameter, for which a default argument is defined. The parameter is an `std::allocator const &`, by default initialized with the default allocator. Objects of the class `match_results` receive their match-related information through their member functions, usually in combination with one of the predefined functions implementing the matching algorithms.

**Member functions**

- **match_results &operator=**:  
  The copy and move assignment operators are available;

- **std::string const &operator[](size_t idx = 0) const**:  
  Returns a (const) reference to sub-match `idx`. By default this is a reference to the full match. If `idx >= size()` (see below) a reference to an empty sub-range of the target string is returned. The behavior of this member is undefined if `ready()` (see below) returns `false`;

- **Iterator begin() const**:  
  Returns an iterator to the first sub-match. `Iterator` is a const-iterator for `const match_results` objects;

- **Iterator cbegin() const**:  
  Returns an iterator to the first sub-match. `Iterator` is a const-iterator;

- **Iterator cend() const**:  
  Returns an iterator pointing beyond the last sub-match. `Iterator` is a const-iterator;

- **Iterator end() const**:  
  Returns an iterator pointing beyond the last sub-match. `Iterator` is a const-iterator for `const match_results` objects;

- **ReturnType format(Parameters) const**:  
  As this member requires a fairly extensive description, it would break the flow of the current overview. Therefore it's covered in the next section (section 18.8.3.1);

- **allocator_type get_allocator() const**:  
  Returns the object's allocator;
• **bool empty() const:**

  Returns **true** if the `match_results` object contains no matches (which is also true after using the default constructor). Otherwise returns **false**;

• **int length(size_t idx = 0) const:**

  Returns the length of sub-match `idx`. By default the length of the full match is returned. If `idx >= size()` (see below) 0 is returned;

• **size_type max_size() const:**

  Returns the maximum number of sub-matches that can be contained in a `match_results` object. This is an implementation dependent constant value;

• **int position(size_t idx = 0) const:**

  Returns the offset of the first character of sub-match `idx`. By default the position of the first character of the full match is returned. If `idx >= size()` (see below) -1 is returned;

• **std::string const &prefix() const:**

  Returns a (const) reference ending at the first character of the full match;

• **bool ready() const:**

  No match results are available from a default constructed `match_results` object. It receives its match results from one of the mentioned matching functions. Returns **true** once the match results are available, and **false** otherwise.

• **size_type size() const:**

  Returns the number of sub-matches. E.g., with a regular expression `(abc)|(def)` and target `defcon` three submatches are reported: the total match (def), the empty text for (abc), and def for the (def) marked sub-expression. Note that when multipliers are used only the last match counts, and is reported. E.g., for the pattern `(a|b)+` and target `aaab` two sub-matches are reported: the total match `aaab`, and the last match (b);

• **std::string str(size_t idx = 0) const:**

  Returns the characters defining sub-match `idx`. By default this is the full match. If `idx >= size()` (see below) an empty string returned;

• **std::string const &suffix() const:**

  Returns a (const) reference starting beyond the last character of the full match;

• **void swap(match_results &other) noexcept:**

  Swaps the current `match_results` object with `other`. Also available as a free function: `void swap(match_results &lhs, match_results &rhs)`, swapping `lhs` and `rhs`. 
18.8. REGULAR EXPRESSIONS

18.8.3.1 The member std::match_results::format

The `match_results::format` member applies the sub-expressions contained in a `match_results` object to the contents of a regular expression format string, producing text in which format specifiers (like `$&`) and escape sequences (like `\n`) are replaced by matching sub-expressions available from the current `match_results` object.

As an initial illustration: if `results` is a `match_results` object and `match[0]` (the fully matched text) equals 'hello world', then calling `format` with the format string `this is <%= results %>` (the format specifier `%=` represents the fully matched text) produces the text `this is [hello world]`.

In the format string all standard C escape sequences are recognized and converted to their usual characters. In addition, the following format specifiers can be used:

- `$'`: corresponds to the text returned by the `prefix` member: all characters in the original target string up to the first character of the fully matched text;
- `$&`: corresponds to the fully matched text;
- `$n`: (n an integral natural number): corresponds to the text returned by the `operator[](n)`;
- `$'`: corresponds to the text returned by the `suffix` member: all characters in the original target string beyond the last character of the fully matched text;
- `$$`: corresponds to the single `$` character.

All overloaded `format` members accept a final `regex_constants::match_flag_type` argument, which is a bit-masked type. All `format` members specify the default argument `match_default`.

The following constants are defined for `match_flag_type` (see also section 18.8.4; `[first, last)` refers to the character sequence being matched):

- `format_default`: (not a bit-mask value, but equal to 0): ECMAScript rules are used to construct strings in `std::regex_replace`;
- `format_first_only`: `std::regex_replace` only replaces the first match;
- `format_no_copy`: non-matching strings are not passed to the output by `std::regex_replace`;
- `format_sed`: POSIX `sed(1)` rules are used to construct strings in `std::regex_replace`;
- `match_any`: if multiple matches are possible, then any match is an acceptable result;
- `match_continuous`: sub-sequences are only matching if they start at `first`;
- `match_not_bol`: the first character in `[first, last)` is treated as an ordinary character: `^` does not match `[first, first)`;
- `match_not_bow`: `\b` does not match `[first, first)`;
- `match_default`: (not a bit-mask value, but equal to 0): ECMAScript rules are used to construct strings in `std::regex_replace`;
- `match_not_eol`: the last character in `[first, last)` is treated as an ordinary character: `$` does not match `[last, last)`;
- `match_not_eow`: `\b` does not match `[last, last)`;
• match_not_null: empty sequences are not considered matches;

• match_prev_avail: -first refers to a valid character position. When specified match_not_bol and match_not_bow are ignored;

Four overloaded versions of the format members are available. Their final regex_constants::match_flag_type parameter, by default initialized to match_default, is not explicitly mentioned below.

The first two overloaded format functions expect an output-iterator which receives the formatted strings. The final output iterator value is returned:

• OutputIter format(OutputIter out, char const *first, char const *last) const:
  
  the characters in the range [first, last) are applied to the sub-expressions stored in the match_results object, and the resulting string is inserted at out (an example is provided at the end of this section);

• OutputIter format(OutputIter out, std::string const &fmt) const:
  
  the contents of fmt are applied to the sub-expressions stored in the match_results object, and the resulting string is inserted at out;

The other two overloaded format functions expect a std::string or NTBS defining the format string, returning a std::string containing the formatted text:

• std::string format(std::string const &fmt) const

• std::string format(char const *fmt) const

Here is an example using various format members. When using the members expecting output iterators you need to make sure that the locations subsequently referred to by the iterators exist. For ostream iterators that should be no problem, but for, e.g., an std::string you must ensure that the string object has a large enough size to accommodate the inserted characters. Of course, the last two overloaded format members elegantly solve that problem. Another possibility is to define your own output iterators (as explained in section 22.14.1). In the example it is assumed that all required headers have been included and that using namespace std has been declared:

```cpp
1: int main()
2: {
3:   regex re("hi");
4:   smatch sm;
5:   
6:   if (not regex_search("just say 'hi, there'", sm, re))
7:     return 1;
8:   
9:   string fmt("fully matched: $&, beyond: "$");
10:  
11:   cout << "Producing a std::string: " << sm.format(fmt) << '\n';
12:  
13:   cout << "Using an ostream_iterator: ";
14:   *sm.format(ostream_iterator<char>(cout), fmt) = '\n';
15: ```
16:    string out(40, 'x');
17:    *sm.format(out.begin(), fmt) = ' ';
18:    cout << "Overwriting an existing std::string: " << out << '
';
19: }

Some details:

- after defining a regex pattern and an smatch (match_results) object the regular expression is matched against a target sentence in line 6;
- in line 9 a format string is defined, using $\&$ and $'$;
- in line 11 the format string is applied to smatch, producing a std::string, which is inserted into cout;
- identical results can be obtained using an iterator. In line 14 an ostream_iterator is used (note that format returns the final ostream_iterator, whose operator* can be used to insert another character into the stream);
- finally, lines 16-18 illustrate how format using a string::iterator can be used to overwrite the contents of an existing std::string object.

18.8.4 Regular expression matching functions

Before using the functions presented in this section the <regex> header file must be included.

We already encountered some uses of regex_search. However, there are more functions implementing regular expression matching algorithms. There are three major families of functions. Each function has a final std::regex_constants::match_flag_type parameter (cf. section 18.8.3.1), which is given the default value std::regex_constants::match_default. This final parameter is not explicitly mentioned below. The three families of functions are:

- bool std::regex_match(Parameters):

  This family of functions is used to match a regular expression against a target text. Only if the regular expression matches the full target text true is returned; otherwise false is returned. Refer to section 18.8.4.1 for an overview of the available overloaded regex_match functions;

- bool std::regex_search(Parameters):

  This family of functions is also used to match a regular expression against a target text. This function returns true once the regular expression matches a sub-string of the target text; otherwise false is returned. See below for an overview of the available overloaded regex_search functions;

- ReturnType std::regex_replace(Parameters):

  This family of functions is used to produce modified texts, using the characters of a target string, a regex object and a format string. This member closely resembles the functionality of the match_results::format member discussed in section 18.8.3.1.
18.8.4.1 Matching target texts: std::regex_match

The regular expression matching function std::regex_match returns true if the regular expression represented by its provided regex argument fully matches the provided target text. The following overloaded variants of this function are available:

- bool regex_match(BidirConstIter first, BidirConstIter last, std::match_results &results, std::regex const &re):
  
  BidirConstIter is a bidirectional const iterator. The range [first, last) defines the target text. The match results are returned in results;

- bool regex_match(BidirConstIter first, BidirConstIter last, std::regex const &re):

  this function behaves like the previous function, but does not return the results of the matching process in a match_results object;

- bool regex_match(char const *target, std::match_results &results, std::regex const &re):

  this function behaves like the first overloaded variant, using the characters in target as its target text;

- bool regex_match(char const *str, std::regex const &re):

  this function behaves like the previous function but does not return the match results;

- bool regex_match(std::string const &target, std::match_results &results, std::regex const &re):

  this function behaves like the first overloaded variant, using the characters in target as its target text;

- bool regex_match(std::string const &str, std::regex const &re):

  this function behaves like the previous function but does not return the match results;

- bool regex_match(std::string const &&, std::match_results &, std::regex &) = delete (the regex_match function does not accept temporary string objects as target strings, as this would result in invalid string iterators in the match_result argument.)

18.8.4.2 Matching sub-strings: std::regex_search

The regular expression matching function std::regex_search returns true if the regular expression represented by its provided regex argument matches a sub-sequence of the provided target text. The following overloaded variants of this function are available:

- bool regex_search(BidirConstIter first, BidirConstIter last, std::match_results &results, std::regex const &re):

  BidirConstIter is a bidirectional const iterator. The range [first, last) defines the target text. The match results are returned in results;

- bool regex_search(BidirConstIter first, BidirConstIter last, std::regex const &re):

  this function behaves like the previous function, but does not return the results of the matching process in a match_results object;
18.8. REGULAR EXPRESSIONS

- bool regex_search(char const *target, std::match_results &results, std::regex const &re):
  this function behaves like the first overloaded variant, using the characters in target as its target text;

- bool regex_search(char const *str, std::regex const &re):
  this function behaves like the previous function but does not return the match results;

- bool regex_search(std::string const &target, std::match_results &results, std::regex const &re):
  this function behaves like the first overloaded variant, using the characters in target as its target text;

- bool regex_search(std::string const &str, std::regex const &re):
  this function behaves like the previous function but does not return the match results;

- bool regex_search(std::string const &&, std::match_results &, std::regex &) = delete:
  the regex_search function does not accept temporary string objects as target strings, as this would result in invalid string iterators in the match_result argument.

18.8.4.3 Modifying target strings: std::regex_replace

The family of std::regex_replace functions uses a regular expression to perform substitution on a sequence of characters. Their functionality closely resembles the functionality of the match_results::format member discussed in section 18.8.3.1. The following overloaded variants are available:

- OutputIt regex_replace(OutputIter out, BidirConstIter first, BidirConstIter last, std::regex const &re, std::string const &fmt):
  OutputIter is an output iterator; BidirConstIter a bidirectional const iterator.
  - The regular expression re is applied to the characters in the range [first, last);
  - For each match, its prefix (starting at the first character for the first match, or just beyond the last character of the previous match) is copied to out, followed by the formatted replacement string defined by fmt.
  - Following the last match, that match result's suffix is copied to out;
  (an example is provided below)

- OutputIt regex_replace( OutputIter out, BidirConstIter first, BidirConstIter last, std::regex const &re, char const *fmt):
  This variant behaves like the first variant;

- std::string regex_replace(std::string const &str, std::regex const &re, std::string const &fmt):
  This variant returns a std::string containing the modified text, and expects a std::string containing the target text. Other than that, it behaves like the first variant;
• `std::string regex_replace(std::string const &str, std::regex const &re, char const *fmt)`:

  This variant behaves like the previous variant;

• `std::string regex_replace(char const *str, std::regex const &re, std::string const &fmt)`:

  This variant behaves like the previous variant;

• `std::string regex_replace(char const *str, std::regex const &re, char const *fmt)`:

  This variant behaves like the previous variant;

The following example embeds all the words in a line of text in a pair of emphasizing html tags (headers and `using namespace std` declaration are assumed):

```cpp
int main()
{
    string target("... oh, when the Saints, ...");

    regex re("\\w+"), re;
    *regex_replace(ostream_iterator<char>(cout),
                   target.cbegin(), target.cend(), re,
                   "<em>$&</em>") = '\n';
}
```

Note that `regex_replace` does not allow conditional replacement. If that’s required then a more complex approach is required, as shown in the next example:

```cpp
1: bool modify(string const &match);
2: size_t replace(string &target, string::iterator begin, string match);
3:
4: int main()
5: {
6:     string target("... oh, when the Saints, ...");
7:     regex re("\\w+"), re;
8:     smatch match;
9:     size_t offset = 0;
10:    while (true)
11:    {
12:        auto begin = target.begin() + offset;
13:        if (not regex_search(string::const_iterator(begin),
14:            target.cend(), match, re))
15:            break;
16:        size_t pos = match.position(0);
17:        offset += pos +
18:        modify(match[0]) ?
19:            replace(target, begin + pos, match[0])
20:            :
21:            match.length(0);
22:    }
```
18.9 Randomization and Statistical Distributions

Before the statistical distributions and accompanying random number generators can be used the `<random>` header file must be included.

The C++11 standard introduces several standard mathematical (statistical) distributions into the STL. These distributions allow programmers to obtain randomly selected values from a selected distribution.

These statistical distributions need to be provided with a random number generating object. Several of such random number generating objects are provided, extending the traditional `rand` function that is part of the C standard library.

These random number generating objects produce pseudo-random numbers, which are then processed by the statistical distribution to obtain values that are randomly selected from the specified distribution.

Although the STL offers various statistical distributions their functionality is fairly limited. The distributions allow us to obtain a random number from these distributions, but probability density functions or cumulative distribution functions are currently not provided by the STL. These functions (distributions as well as the density and the cumulative distribution functions) are, however, available in other libraries, like the boost math library\(^4\) (specifically: http://www.boost.org/doc/libs/1_44_0/libs/math/doc/sf_and_dist/html/index.html).

It is beyond the scope of the C++ Annotations to discuss the mathematical characteristics of the various distributions that are supported by the C++11 standard. The interested reader is referred to the pertinent mathematical textbooks (like Stuart and Ord's (2009) *Kendall's Advanced Theory of Statistics*, Wiley) or to web-locations like http://en.wikipedia.org/wiki/Bernoulli_distribution.

\(^4\)http://www.boost.org/
18.9.1 Random Number Generators

The following generators are available:

<table>
<thead>
<tr>
<th>Class template</th>
<th>Integral/Floating point</th>
<th>Quality</th>
<th>Speed</th>
<th>Size of state</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear_congruential_engine</td>
<td>Integral</td>
<td>Medium</td>
<td>Medium</td>
<td>1</td>
</tr>
<tr>
<td>subtract_with_carry_engine</td>
<td>Both</td>
<td>Medium</td>
<td>Fast</td>
<td>25</td>
</tr>
<tr>
<td>mersenne_twister_engine</td>
<td>Integral</td>
<td>Good</td>
<td>Fast</td>
<td>624</td>
</tr>
</tbody>
</table>

The `linear_congruential_engine` random number generator computes

\[ \text{value}_{i+1} = (OA \times \text{value}_i + c) \mod m \]

It expects template arguments for, respectively, the data type to contain the generated random values; the multiplier `a`; the additive constant `c`; and the modulo value `m`. Example:

```cpp
linear_congruential_engine<int, 10, 3, 13> lincon;
```

The `linear_congruential_engine` generator may be seeded by providing its constructor with a seeding-argument. E.g., `lincon(time(0))`.

The `subtract_with_carry_engine` random number generator computes

\[ \text{value}_i = (\text{value}_{i-s} - \text{value}_{i-r} - \text{carry}_{i-1}) \mod m \]

It expects template arguments for, respectively, the data type to contain the generated random values; the modulo value `m`; and the subtractive constants `s` and `r`. Example:

```cpp
subtract_with_carry_engine<int, 13, 3, 13> subcar;
```

The `subtract_with_carry_engine` generator may be seeded by providing its constructor with a seeding-argument. E.g., `subcar(time(0))`.

The predefined `mersenne_twister_engine mt19937` (predefined using a typedef defined by the `<random>` header file) is used in the examples below. It can be constructed using `mt19937 mt` or it can be seeded by providing its constructor with an argument (e.g., `mt19937 mt(time(0))`). Other ways to initialize the `mersenne_twister_engine` are beyond the scope of the C++ Annotations (but see Lewis et al.\(^5\) (1969)).

The random number generators may also be seeded by calling their members `seed` accepting unsigned long values or generator functions (as in `lc.seed(time(0))`, `lc.seed(mt)`).

The random number generators offer members `min` and `max` returning, respectively, their minimum and maximum values (inclusive). If a reduced range is required the generators can be nested in a function or class adapting the range.

18.9.2 Statistical distributions

In the following sections the various statistical distributions that are supported by the C++11 standard are covered. The notation `RNG` is used to indicate a Random Number Generator and

**18.9. RANDOMIZATION AND STATISTICAL DISTRIBUTIONS**

URING is used to indicate a *Uniform Random Number Generator*. With each distribution a struct `param_type` is defined containing the distribution's parameters. The organization of these `param_type` structs depends on (and is described at) the actual distribution.

All distributions offer the following members (`result_type` refers to the type name of the values returned by the distribution):

- `result_type max() const`
  returns the distribution's least upper bound;

- `result_type min() const`
  returns the distribution's greatest lower bound;

- `param_type param() const`
  returns the object's `param_type` struct;

- `void param(const param_type &param)`
  redefines the parameters of the distribution;

- `void reset()`
  clears all of its cached values;

All distributions support the following operators (`distribution-name` should be replaced by the name of the intended distribution, e.g., `normal_distribution`):

- `template<typename URNG> result_type operator()(URNG &urng)`
  returns the next random value from the statistical distribution, with the function object `urng` returning the next random number selected from a uniform random distribution;

- `template<typename URNG> result_type operator()(URNG &urng, param_type &param)`
  returns the next random value from the statistical distribution initialized with the parameters provided by the `param` struct. The function object `urng` returns the next random number selected from a uniform random distribution;

- `std::istream &operator>>(std::istream &in, distribution-name &object)`:
  The parameters of the distribution are extracted from an `std::istream`;

- `std::ostream &operator<<(std::ostream &out, distribution-name const &bd)`:
  The parameters of the distribution are inserted into an `std::ostream`;

The following example shows how the distributions can be used. Replacing the name of the distribution (*normal_distribution*) by another distribution's name is all that is required to switch distributions. All distributions have parameters, like the mean and standard deviation of the normal distribution, and all parameters have default values. The names of the parameters vary over distributions and are mentioned below at the individual distributions. Distributions offer members returning or setting their parameters.

Most distributions are defined as class templates, requiring the specification of a data type that is used for the function's return type. If so, an empty template parameter type specification (`<>`) will get you the default type. The default types are either `double` (for real valued return types) or `int` (for integral valued return types). The template parameter type specification must be omitted with distributions that are not defined as template classes.

Here is an example showing the use of the statistical distributions, applied to the normal distribution:

```cpp
#include <iostream>
```
#include <ctime>
#include <random>
using namespace std;

int main()
{
    std::mt19937 engine(time(0));
    std::normal_distribution<> dist;

    for (size_t idx = 0; idx < 10; ++idx)
        std::cout << "a random value: " << dist(engine) << "\n";
    cout << '
' << dist.min() << " " << dist.max() << '
';
}

18.9.2.1 Bernoulli distribution

The bernoulli_distribution is used to generate logical truth (boolean) values with a certain probability p. It is equal to a binomial distribution for one experiment (cf 18.9.2.2).

The bernoulli distribution is not defined as a class template.

Defined types:

typedef bool result_type;
struct param_type
{
    explicit param_type(double prob = 0.5);
    double p() const; // returns prob
};

Constructor and members:

- bernoulli_distribution(double prob = 0.5) constructs a bernoulli distribution with probability prob of returning true;

- double p() const returns prob;

- result_type min() const returns false;

- result_type max() const returns true;

18.9.2.2 Binomial distribution

The binomial_distribution<IntType = int> is used to determine the probability of the number of successes in a sequence of n independent success/failure experiments, each of which yields success with probability p.

The template type parameter IntType defines the type of the generated random value, which must be an integral type.
18.9. RANDOMIZATION AND STATISTICAL DISTRIBUTIONS

Defined types:

```cpp
typedef IntType result_type;
struct param_type
{
    explicit param_type(IntType trials, double prob = 0.5);
    IntType t() const; // returns trials
    double p() const; // returns prob
};
```

Constructors and members and example:

- `binomial_distribution<>(IntType trials = 1, double prob = 0.5)` constructs a binomial distribution for trials experiments, each having probability prob of success.
- `binomial_distribution<>(param_type const &param)` constructs a binomial distribution according to the values stored in the param struct.
- IntType t() const returns trials;
- double p() const returns prob;
- result_type min() const returns 0;
- result_type max() const returns trials;

18.9.2.3 Cauchy distribution

The `cauchy_distribution<RealType = double>` looks similar to a normal distribution. But cauchy distributions have heavier tails. When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of how sensitive the tests are to heavy-tail departures from normality.

The mean and standard deviation of the Cauchy distribution are undefined.

Defined types:

```cpp
typedef RealType result_type;
struct param_type
{
    explicit param_type(RealType a = RealType(0),
                        RealType b = RealType(1));
    double a() const;
    double b() const;
};
```

Constructors and members:

- `cauchy_distribution<>(RealType a = RealType(0), RealType b = RealType(1))` constructs a cauchy distribution with specified a and b parameters.
• cauchy_distribution<param_type const &param> constructs a cauchy distribution according to the values stored in the param struct.

• RealType a() const returns the distribution’s a parameter;

• RealType b() const returns the distribution’s b parameter;

• result_type min() const returns the smallest positive result_type value;

• result_type max() const returns the maximum value of result_type;

18.9.2.4 Chi-squared distribution

The chi_squared_distribution<RealType = double> with \( n \) degrees of freedom is the distribution of a sum of the squares of \( n \) independent standard normal random variables.

Note that even though the distribution’s parameter \( n \) usually is an integral value, it doesn’t have to be integral, as the chi_squared distribution is defined in terms of functions (\texttt{exp} and \texttt{Gamma}) that take real arguments (see, e.g., the formula shown in the \texttt{<bits/random.h>} header file, provided with the Gnu g++ compiler distribution).

The chi-squared distribution is used, e.g., when testing the goodness of fit of an observed distribution to a theoretical one.

Defined types:

\[
\text{typedef RealType result_type;}
\]

\[
\text{struct param_type}
\]

\[
\begin{align*}
\text{explicit param_type(RealType n = RealType(1));} \\
\text{RealType n() const;}
\end{align*}
\]

Constructors and members:

• chi_squared_distribution<RealType = double>\((\text{RealType n = 1})\) constructs a chi_squared distribution with specified number of degrees of freedom.

• chi_squared_distribution<param_type const &param> constructs a chi_squared distribution according to the value stored in the param struct;

• IntType n() const returns the distribution’s degrees of freedom;

• result_type min() const returns 0;

• result_type max() const returns the maximum value of result_type;
18.9. RANDOMIZATION AND STATISTICAL DISTRIBUTIONS

18.9.2.5 Extreme value distribution

The `extreme_value_distribution<RealType = double>` is related to the Weibull distribution and is used in statistical models where the variable of interest is the minimum of many random factors, all of which can take positive or negative values.

It has two parameters: a location parameter $a$ and scale parameter $b$. See also http://www.itl.nist.gov/div898/handbook/apr/section1/apr163.htm

Defined types:

```cpp
typedef RealType result_type;

struct param_type
{
    explicit param_type(RealType a = RealType(0),
                        RealType b = RealType(1));

    RealType a() const; // the location parameter
    RealType b() const; // the scale parameter
};
```

Constructors and members:

- `extreme_value_distribution<RealType = double>(RealType a = 0, RealType b = 1)` constructs an extreme value distribution with specified $a$ and $b$ parameters;
- `extreme_value_distribution<RealType = double>(param_type const &param)` constructs an extreme value distribution according to the values stored in the `param` struct.
- `RealType a() const` returns the distribution's location parameter;
- `RealType stddev() const` returns the distribution's scale parameter;
- `result_type min() const` returns the smallest positive value of `result_type`;
- `result_type max() const` returns the maximum value of `result_type`;

18.9.2.6 Exponential distribution

The `exponential_distribution<RealType = double>` is used to describe the lengths between events that can be modelled with a homogeneous Poisson process. It can be interpreted as the continuous form of the geometric distribution.

Its parameter $\text{prob}$ defines the distribution's $\lambda$ parameter, called its rate parameter. Its expected value and standard deviation are both $1 / \lambda$.

Defined types:

```cpp
typedef RealType result_type;
```
struct param_type
{
  explicit param_type(RealType lambda = RealType(1))
  {
    RealType lambda() const;
  };
}

Constructors and members:

- exponential_distribution<>(RealType lambda = 1) constructs an exponential distribution with specified lambda parameter.
- exponential_distribution<>(param_type const &param) constructs an exponential distribution according to the value stored in the param struct.
- RealType lambda() const returns the distribution's lambda parameter;
- result_type min() const returns 0;
- result_type max() const returns the maximum value of result_type;

18.9.2.7 Fisher F distribution

The fisher_f_distribution<RealType = double> is intensively used in statistical methods like the Analysis of Variance. It is the distribution resulting from dividing two Chi-squared distributions.

It is characterized by two parameters, being the degrees of freedom of the two chi-squared distributions.

Note that even though the distribution's parameter \( n \) usually is an integral value, it doesn't have to be integral, as the Fisher F distribution is constructed from Chi-squared distributions that accept a non-integral parameter value (see also section 18.9.2.4).

Defined types:

typedef RealType result_type;

struct param_type
{
  explicit param_type(RealType m = RealType(1),
                      RealType n = RealType(1));

  RealType m() const; // The degrees of freedom of the nominator
  RealType n() const; // The degrees of freedom of the denominator
};

Constructors and members:

- fisher_f_distribution<>(RealType m = RealType(1), RealType n = RealType(1)) constructs a fisher_f distribution with specified degrees of freedom.
18.9. RANDOMIZATION AND STATISTICAL DISTRIBUTIONS

- `fisher_f_distribution<>(param_type const &param)` constructs a `fisher_f` distribution according to the values stored in the `param` struct.

- `RealType m() const` returns the degrees of freedom of the nominator;

- `RealType n() const` returns the degrees of freedom of the denominator;

- `result_type min() const` returns 0;

- `result_type max() const` returns the maximum value of `result_type`;

18.9.2.8 Gamma distribution

The `gamma_distribution<RealType = double>` is used when working with data that are not distributed according to the normal distribution. It is often used to model waiting times.

It has two parameters, `alpha` and `beta`. Its expected value is `alpha * beta` and its standard deviation is `alpha * beta^2`.

Defined types:

```cpp
typedef RealType result_type;

struct param_type
{
    explicit param_type(RealType alpha = RealType(1),
                        RealType beta = RealType(1));

    RealType alpha() const;
    RealType beta() const;
};
```

Constructors and members:

- `gamma_distribution<>(RealType alpha = 1, RealType beta = 1)` constructs a gamma distribution with specified `alpha` and `beta` parameters.

- `gamma_distribution<>(param_type const &param)` constructs a gamma distribution according to the values stored in the `param` struct.

- `RealType alpha() const` returns the distribution's alpha parameter;

- `RealType beta() const` returns the distribution's beta parameter;

- `result_type min() const` returns 0;

- `result_type max() const` returns the maximum value of `result_type`;
18.9.2.9 Geometric distribution

The geometric_distribution<IntType = int> is used to model the number of bernoulli trials (cf. 18.9.2.1) needed until the first success.

It has one parameter, prob, representing the probability of success in an individual bernoulli trial.

Defined types:

```
typedef IntType result_type;

struct param_type
{
    explicit param_type(double prob = 0.5);
    double p() const;
};
```

Constructors, members and example:

- geometric_distribution<>(double prob = 0.5) constructs a geometric distribution for bernoulli trials each having probability prob of success.
- geometric_distribution<>(param_type const &param) constructs a geometric distribution according to the values stored in the param struct.
- double p() const returns the distribution’s prob parameter;
- param_type param() const returns the object’s param_type structure;
- void param(const param_type &param) redefines the parameters of the distribution;
- result_type min() const returns the distribution’s lower bound (= 0);
- result_type max() const returns the distribution’s upper bound;
- template<typename URNG> result_type operator()(URNG &urng) returns the next random value from the geometric distribution
- template<typename URNG> result_type operator()(URNG &urng, param_type &param) returns the next random value from a geometric distribution initialized by the provided param struct.
- The random number generator that is passed to the generating functions must return integral values. Here is an example showing how the geometric distribution can be used:

```c++
#include <iostream>
#include <ctime>
#include <random>

int main()
{
    std::linear_congruential_engine<unsigned, 7, 3, 61> engine(0);
```
18.9. RANDOMIZATION AND STATISTICAL DISTRIBUTIONS

```cpp
class std::geometric_distribution<>
{  
    std::cout << "a random value: " << dist(engine) << "\n";
    std::cout << dist.min() << " " << dist.max() << "\n";
};
```

18.9.2.10 Log-normal distribution

The `lognormal_distribution<RealType = double>` is a probability distribution of a random variable whose logarithm is normally distributed. If a random variable $X$ has a normal distribution, then $Y = e^X$ has a log-normal distribution.

It has two parameters, $m$ and $s$ representing, respectively, the mean and standard deviation of $\ln(X)$.

Defined types:

```cpp
typedef RealType result_type;

struct param_type
{
    explicit param_type(RealType m = RealType(0),
                        RealType s = RealType(1));
    RealType m() const;
    RealType s() const;
};
```

Constructor and members:

- `lognormal_distribution<RealType = double>(RealType m = 0, RealType s = 1)` constructs a log-normal distribution for a random variable whose mean and standard deviation is, respectively, $m$ and $s$.
- `lognormal_distribution<RealType = double>(param_type const &param)` constructs a log-normal distribution according to the values stored in the `param` struct.
- `RealType m() const` returns the distribution's $m$ parameter;
- `RealType stddev() const` returns the distribution's $s$ parameter;
- `result_type min() const` returns 0;
- `result_type max() const` returns the maximum value of `result_type`;

18.9.2.11 Normal distribution

The normal_distribution<RealType = double> is commonly used in science to describe complex phenomena. When predicting or measuring variables, errors are commonly assumed to be normally distributed.

It has two parameters, mean and standard deviation.

Defined types:

```cpp
typedef RealType result_type;

struct param_type
{
    explicit param_type(RealType mean = RealType(0),
                        RealType stddev = RealType(1));

    RealType mean() const;
    RealType stddev() const;
};
```

Constructors and members:

- normal_distribution<RealType mean = 0, RealType stddev = 1> constructs a normal distribution with specified mean and stddev parameters. The default parameter values define the standard normal distribution;
- normal_distribution<>(param_type const &param) constructs a normal distribution according to the values stored in the param struct.
- RealType mean() const returns the distribution's mean parameter;
- RealType stddev() const returns the distribution's stddev parameter;
- result_type min() const returns the lowest positive value of result_type;
- result_type max() const returns the maximum value of result_type;

18.9.2.12 Negative binomial distribution

The negative_binomial_distribution<IntType = int> probability distribution describes the number of successes in a sequence of Bernoulli trials before a specified number of failures occurs. For example, if one throws a die repeatedly until the third time 1 appears, then the probability distribution of the number of other faces that have appeared is a negative binomial distribution.

It has two parameters: (IntType) k (> 0), being the number of failures until the experiment is stopped and (double) p the probability of success in each individual experiment.

Defined types:

```cpp
typedef IntType result_type;
```
**18.9. RANDOMIZATION AND STATISTICAL DISTRIBUTIONS**

```cpp
struct param_type
{
    explicit param_type(IntType k = IntType(1), double p = 0.5);
    IntType k() const;
    double p() const;
};
```

Constructors and members:

- `negative_binomial_distribution<>(IntType k = IntType(1), double p = 0.5)` constructs a negative_binomial distribution with specified \( k \) and \( p \) parameters;
- `negative_binomial_distribution<>(param_type const &param)` constructs a negative_binomial distribution according to the values stored in the `param` struct.
- `IntType k() const` returns the distribution’s \( k \) parameter;
- `double p() const` returns the distribution’s \( p \) parameter;
- `result_type min() const` returns 0;
- `result_type max() const` returns the maximum value of `result_type`;

### 18.9.2.13 Poisson distribution

The `poisson_distribution<IntType = int>` is used to model the probability of a number of events occurring in a fixed period of time if these events occur with a known probability and independently of the time since the last event.

It has one parameter, \( \text{mean} \), specifying the expected number of events in the interval under consideration. E.g., if on average 2 events are observed in a one-minute interval and the duration of the interval under study is 10 minutes then \( \text{mean} = 20 \).

Defined types:

```cpp
typedef IntType result_type;

struct param_type
{
    explicit param_type(double mean = 1.0);
    double mean() const;
};
```

Constructors and members:

- `poisson_distribution<>(double mean = 1)` constructs a poisson distribution with specified `mean` parameter.
poisson_distribution<>(param_type const &param) constructs a poisson distribution according to the values stored in the param struct.

double mean() const returns the distribution's mean parameter;

result_type min() const returns 0;

result_type max() const returns the maximum value of result_type;

18.9.2.14 Student t distribution

The student_t_distribution<RealType = double> is a probability distribution that is used when estimating the mean of a normally distributed population from small sample sizes.

It is characterized by one parameter: the degrees of freedom, which is equal to the sample size - 1.

Defined types:

typedef RealType result_type;

struct param_type
{
    explicit param_type(RealType n = RealType(1));

    RealType n() const;  // The degrees of freedom
};

Constructors and members:

student_t_distribution<>(RealType n = RealType(1)) constructs a student_t distribution with indicated degrees of freedom.

student_t_distribution<>(param_type const &param) constructs a student_t distribution according to the values stored in the param struct.

RealType n() const returns the degrees of freedom;

result_type min() const returns 0;

result_type max() const returns the maximum value of result_type;

18.9.2.15 Uniform int distribution

The uniform_int_distribution<IntType = int> can be used to select integral values randomly from a range of uniformly distributed integral values.

It has two parameters, a and b, specifying, respectively, the lowest value that can be returned and the highest value that can be returned.
18.9. RANDOMIZATION AND STATISTICAL DISTRIBUTIONS

Defined types:

typedef IntType result_type;

struct param_type
{
    explicit param_type(IntType a = 0, IntType b = max(IntType));

    IntType a() const;
    IntType b() const;
};

Constructors and members:

• uniform_int_distribution<>(IntType a = 0, IntType b = max(IntType)) constructs a uniform_int distribution for the specified range of values.

• uniform_int_distribution<>(param_type const &param) constructs a uniform_int distribution according to the values stored in the param struct.

• IntType a() const returns the distribution’s a parameter;

• IntType b() const returns the distribution’s b parameter;

• result_type min() const returns the distribution’s a parameter;

• result_type max() const returns the distribution’s b parameter;

18.9.2.16 Uniform real distribution

The uniform_real_distribution<RealType = double> can be used to select RealType values randomly from a range of uniformly distributed RealType values.

It has two parameters, a and b, specifying, respectively, the half-open range of values ([a, b)) that can be returned by the distribution.

Defined types:

typedef RealType result_type;

struct param_type
{
    explicit param_type(RealType a = 0, RealType b = max(RealType));

    RealType a() const;
    RealType b() const;
};

Constructors and members:

• uniform_real_distribution<>(RealType a = 0, RealType b = max(RealType)) constructs a uniform_real distribution for the specified range of values.
• uniform_real_distribution<>::(param_type const &param) constructs a uniform_real
distribution according to the values stored in the param struct.

• RealType a() const
  returns the distribution’s a parameter;

• RealType b() const
  returns the distribution’s b parameter;

• result_type min() const
  returns the distribution’s a parameter;

• result_type max() const
  returns the distribution’s b parameter;

18.9.2.17 Weibull distribution

The weibull_distribution<RealType = double> is commonly used in reliability engineering
and in survival (life data) analysis.

It has two or three parameters and the two-parameter variant is offered by the STL. The three
parameter variant has a shape (or slope) parameter, a scale parameter and a location parameter.
The two parameter variant implicitly uses the location parameter value 0. In the two parameter
variant the shape parameter (a) and the scale parameter (b) are provided. See
http://www.weibull.com/hotwire/issue14/relbasics14.htm for an interesting coverage of
the meaning of the Weibull distribution’s parameters.

Defined types:

typedef RealType result_type;

struct param_type
{
  explicit param_type(RealType a = RealType(1),
                     RealType b = RealType(1));

  RealType a() const; // the shape (slope) parameter
  RealType b() const; // the scale parameter
};

Constructors and members:

• weibull_distribution<>::(RealType a = 1, RealType b = 1) constructs a weibull
distribution with specified a and b parameters;

• weibull_distribution<>::(param_type const &param) constructs a weibull distribution
according to the values stored in the param struct.

• RealType a() const
  returns the distribution’s shape (or slope) parameter;

• RealType stddev() const
  returns the distribution’s scale parameter;

• result_type min() const
  returns 0;
• `result_type max() const` returns the maximum value of `result_type;`
Chapter 19

The STL Generic Algorithms

19.1 The Generic Algorithms

Before using the generic algorithms presented in this chapter, except for those in the Operators category (defined below), the <algorithm> header file must be included. Before using a generic algorithm in the Operators category the <numeric> header file must be included.

In the previous chapter the Standard Template Library (STL) was introduced. An important element of the STL, the generic algorithms, was not covered in that chapter as they form a fairly extensive part of the STL. Over time the STL has grown considerably, mainly as a result of a growing importance and appreciation of templates. Covering generic algorithm in the STL chapter itself would turn that chapter into an unwieldy one and so the generic algorithms were moved to a chapter of their own.

Generic algorithms perform an amazing task. Due to the strength of templates, algorithms could be developed that can be applied to a wide range of different data types while maintaining type safety. The prototypical example of this is the sort generic algorithm. To contrast: while C requires programmers to write callback functions in which type-unsafe void const * parameters have to be used, internally forcing the programmer to resort to casts, STL's sort frequently allows the programmer merely to state something akin to

\[
\text{sort(first-element, last-element)}
\]

Generic algorithms should be used wherever possible. Avoid the urge to design your own code for commonly encountered algorithms. Make it a habit to first thoroughly search the generic algorithms for an available candidate. The generic algorithms should become your weapon of choice when writing code: acquire full familiarity with them and make their use your 'second nature'.

This chapter's sections cover the STL's generic algorithms in alphabetical order. For each algorithm the following information is provided:

- The required header file;
- The function prototype;
- A short description;
- A short example.
In the prototypes of the algorithms `Type` is used to specify a generic data type. Furthermore, the particular type of iterator (see section 18.2) that is required is mentioned as well as other generic types that might be required (e.g., performing `BinaryOperations`, like `plus<Type>`). Although iterators are commonly provided by abstract containers and comparable pre-defined data structures, at some point you may want to design your own iterators. Section 22.14 offers guidelines for constructing your own iterator classes and provides an overview of operators that must be implemented for the various types of iterators.

Almost every generic algorithm expects an iterator range `[first, last)`, defining the series of elements on which the algorithm operates. The iterators point to objects or values. When an iterator points to a `Type` value or object, function objects used by the algorithms usually receive `Type const &` objects or values. Usually function objects cannot modify the objects they receive as their arguments. This does not hold true for modifying generic algorithms, which are of course able to modify the objects they operate upon.

Generic algorithms may be categorized. The C++ Annotations distinguishes the following categories of generic algorithms:

- Comparators: comparing (ranges of) elements:
  - `equal`; `includes`; `lexicographical_compare`; `max`; `min`; `mismatch`;
- Copiers: performing copy operations:
  - `copy`; `copy_backward`; `partial_sort_copy`; `remove_copy`; `remove_copy_if`; `replace_copy`;
  - `replace_copy_if`; `reverse_copy`; `rotate_copy`; `unique_copy`;
- Counters: performing count operations:
  - `count`; `count_if`;
- Heap operators: manipulating a max-heap:
  - `make_heap`; `pop_heap`; `push_heap`; `sort_heap`;
- Initializers: initializing data:
  - `fill`; `fill_n`; `generate`; `generate_n`;
- Operators: performing arithmetic operations of some sort:
  - `accumulate`; `adjacent_difference`; `inner_product`; `partial_sum`;
- Searchers: performing search (and find) operations:
  - `adjacent_find`; `binary_search`; `equal_range`; `find`; `find_end`; `find_first_of`; `find_if`;
  - `lower_bound`; `max_element`; `min_element`; `search`; `search_n`; `set_difference`;
  - `set_intersection`;
  - `set_symmetric_difference`; `set_union`; `upper_bound`;
- Shufflers: performing reordering operations (sorting, merging, permuting, shuffling, swapping):
  - `inplace_merge`; `iter_swap`; `merge`; `next_permutation`; `nth_element`; `partial_sort`;
  - `partial_sort_copy`; `partition`; `prev_permutation`; `random_shuffle`; `remove`; `remove_copy`;
  - `remove_copy_if`; `remove_if`; `reverse`; `reverse_copy`; `rotate`; `rotate_copy`; `sort`;
  - `stable_partition`; `stable_sort`; `swap`; `unique`;
- Visitors: visiting elements in a range:
  - `for_each`; `replace`; `replace_copy`; `replace_copy_if`; `replace_if`; `transform`; `unique_copy`;
19.1. THE GENERIC ALGORITHMS

19.1.1 accumulate

- **Header file:** `<numeric>`

- **Function prototypes:**
  - Type accumulate(InputIterator first, InputIterator last, Type init);
  - Type accumulate(InputIterator first, InputIterator last, Type init, BinaryOperation op);

- **Description:**
  - The first prototype: `operator+` is applied to all elements implied by the iterator range and to the initial value `init`. The resulting value is returned.
  - The second prototype: the binary operator `op` is applied to all elements implied by the iterator range and to the initial value `init`, and the resulting value is returned.

- **Example:**

```cpp
#include <numeric>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
    int ia[] = {1, 2, 3, 4};
    vector<int> iv(ia, ia + 4);

    cout <<
        "Sum of values: " << accumulate(iv.begin(), iv.end(), int()) << 
        "\n"
        "Product of values: " << accumulate(iv.begin(), iv.end(), int()), multilies<int>()) << 'n';
} /*
 * Displays:
 * Sum of values: 10
 * Product of values: 24
 */
```

19.1.2 adjacent_difference

- **Header file:** `<numeric>`

- **Function prototypes:**
  - OutputIterator adjacent_difference(InputIterator first, InputIterator last, OutputIterator result);
  - OutputIterator adjacent_difference(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation op);
• Description: All operations are performed on the original values, all computed values are returned values.

- The first prototype: the first returned element is equal to the first element of the input range. The remaining returned elements are equal to the difference of the corresponding element in the input range and its previous element.

- The second prototype: the first returned element is equal to the first element of the input range. The remaining returned elements are equal to the result of the binary operator \( \text{op} \) applied to the corresponding element in the input range (left operand) and its previous element (right operand).

• Example:

```cpp
#include <numeric>
#include <vector>
#include <iterator>
#include <iostream>

using namespace std;

int main()
{
    int ia[] = {1, 2, 5, 10};
    vector<int> iv(ia, ia + 4);
    vector<int> ov(iv.size());

    adjacent_difference(iv.begin(), iv.end(), ov.begin());

    copy(ov.begin(), ov.end(), ostream_iterator<int>(cout, " "));
    cout << '\n';

    adjacent_difference(iv.begin(), iv.end(), ov.begin(), minus<int>());

    copy(ov.begin(), ov.end(), ostream_iterator<int>(cout, " "));
    cout << '\n';

    /* Displays:
       1 1 3 5
       1 1 3 5
    */
}
```

19.1.3 adjacent_find

• Header file: <algorithm>

• Function prototypes:

- ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);
- OutputIterator adjacent_find(ForwardIterator first, ForwardIterator last, Predicate pred);

• Description:

- The first prototype: the iterator pointing to the first element of the first pair of two adjacent equal elements is returned. If no such element exists, \( \text{last} \) is returned.
The second prototype: the iterator pointing to the first element of the first pair of two adjacent elements for which the binary predicate `pred` returns `true` is returned. If no such element exists, `last` is returned.

- Example:

```cpp
#include <algorithm>
#include <string>
#include <iostream>
using namespace std;

class SquaresDiff
{
    size_t d_minimum;
public:
    SquaresDiff(size_t minimum)
    :
        d_minimum(minimum)
    {}
    bool operator()(size_t first, size_t second)
    {
        return second * second - first * first >= d_minimum;
    }
};

int main()
{
    string sarr[] = {
        "Alpha", "bravo", "charley", "delta", "echo", "echo", "foxtrot", "golf"
    };  
    string *last = sarr + sizeof(sarr) / sizeof(string);
    string *result = adjacent_find(sarr, last);
    cout << *result << '\n';
    result = adjacent_find(++result, last);
    cout << "Second time, starting from the next position:\n" <<
    {
        result == last ?
            "** No more adjacent equal elements **" :
            "*result"
    } << '\n';
    size_t iv[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    size_t *ilast = iv + sizeof(iv) / sizeof(size_t);
    size_t *ires = adjacent_find(iv, ilast, SquaresDiff(10));
    cout <<
        "The first numbers for which the squares differ at least 10: "
        << *ires << " and " << *(ires + 1) << '\n';
    /*
     Displays:
     */
CHAPTER 19. THE STL GENERIC ALGORITHMS

```cpp
#include <algorithm>
#include <string>
#include <iostream>
#include <functional>
using namespace std;

int main()
{
    string sarr[] =
    {
        "alpha", "bravo", "charley", "delta", "echo",
        "foxtrot", "golf", "hotel"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);
    bool result = binary_search(sarr, last, "foxtrot");
    cout << (result ? "found " : "didn’t find ") << "foxtrot" << ‘\n’;
    reverse(sarr, last); // reverse the order of elements
    result = binary_search(sarr, last, "foxtrot");
    cout << (result ? "found " : "didn’t find ") << "foxtrot" << ‘\n’;
    // ok when using appropriate comparator:
    result = binary_search(sarr, last, "foxtrot", Comparator comp);
    cout << (result ? "found " : "didn’t find ") << "foxtrot" << ‘\n’;
```

19.1.4 binary_search

- **Header file:** `<algorithm>`
- **Function prototypes:**
  - `bool binary_search(ForwardIterator first, ForwardIterator last, Type const &value);`
  - `bool binary_search(ForwardIterator first, ForwardIterator last, Type const &value, Comparator comp);`
- **Description:**
  - The first prototype: value is looked up using binary search in the series of elements implied by the iterator range `[first, last)`. The elements in the range must have been sorted by the `Type::operator<` function. True is returned if the element was found, false otherwise.
  - The second prototype: value is looked up using binary search in the series of elements implied by the iterator range `[first, last)`. The elements in the range must have been sorted by the Comparator function object. True is returned if the element was found, false otherwise.
- **Example:**
19.1.5 copy

- **Header file:** `<algorithm>`
- **Function prototype:**
  
  - `OutputIterator copy(InputIterator first, InputIterator last, OutputIterator destination);`
- **Description:**
  
  - The series of elements implied by the iterator range `[first, last)` is copied to an output range, starting at `destination` using the assignment operator of the underlying data type. The return value is the `OutputIterator` pointing just beyond the last element that was copied to the destination range (so, ‘last’ in the destination range is returned).
- **Example:**
  
  Note the second call to `copy`. It uses an `ostream_iterator` for string objects. This iterator writes the string values to the specified `ostream` (i.e., `cout`), separating the values by the specified separation string (i.e., " ").

```cpp
#include <algorithm>
#include <string>
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
    string sarr[] = {
        "alpha", "bravo", "charley", "delta", "echo",
        "foxtrot", "golf", "hotel"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);
    copy(sarr + 2, last, sarr); // move all elements two positions left

    // copy to cout using an ostream_iterator
    // for strings,
    copy(sarr, last, ostream_iterator<string>(cout, " "));
    cout << '\n';
}
// Displays: charley delta echo foxtrot golf hotel
• See also: unique_copy

19.1.6 copy_backward

• Header file: <algorithm>

• Function prototype:

  ```
  – BidirectionalIterator copy_backward(InputIterator first,
    InputIterator last, BidirectionalIterator last2);
  ```

• Description:

  The series of elements implied by the iterator range \([first, last)\) are copied from
  the element at position \(last - 1\) until (and including) the element at position \(first\) to
  the element range, ending at position \(last2 - 1\) using the assignment operator of the
  underlying data type. The destination range is therefore \([last2 - (last - first),
  last2)\).

  Note that this algorithm does not reverse the order of the elements when copying them to
  the destination range.

  The return value is the BidirectionalIterator pointing to the last element that was copied
  to the destination range (so, ‘first’ in the destination range, pointed to by \(last2 - (last
  - first)\), is returned).

• Example:

  ```
  #include <algorithm>
  #include <string>
  #include <iostream>
  using namespace std;
  
  int main()
  {
    string sarr[] =
    {
      "alpha", "bravo", "charley", "delta", "echo",
      "foxtrot", "golf", "hotel"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);
    copy
    (       
      copy_backward(sarr + 3, last, last - 3),
      last,
      ostream_iterator<string>(cout, " ")
    );
    cout << ‘\n’;
  }
  // Displays: golf hotel foxtrot golf hotel foxtrot golf hotel
  ```

19.1.7 count

• Header file: <algorithm>
19.1. THE GENERIC ALGORITHMS

• Function prototype:
  - `size_t count(InputIterator first, InputIterator last, Type const &value);`

• Description:
  - The number of times `value` occurs in the iterator range `[first, last)` is returned. Uses `Type::operator==` to determine whether `value` is equal to an element in the iterator range.

• Example:

```cpp
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
    int ia[] = {1, 2, 3, 4, 3, 4, 2, 1, 3};

    cout << "Number of times the value 3 is available: " << count(ia, ia + sizeof(ia) / sizeof(int), 3) << '
';
}
```

// Displays: Number of times the value 3 is available: 3

19.1.8 count_if

• Header file: `<algorithm>`

• Function prototype:
  - `size_t count_if(InputIterator first, InputIterator last, Predicate predicate);`

• Description:
  - The number of times unary predicate `predicate` returns `true` when applied to the elements implied by the iterator range `[first, last)` is returned.

• Example:

```cpp
#include <algorithm>
#include <iostream>
using namespace std;

class Odd
{
    public:
        bool operator()(int value) const
        {
            return value & 1;
        }
};

int main()
```
{ int ia[] = {1, 2, 3, 4, 3, 4, 2, 1, 3};
    cout << "The number of odd values in the array is: " <<
    count_if(ia, ia + sizeof(ia) / sizeof(int), Odd()) << '\n';
} // Displays: The number of odd values in the array is: 5

19.1.9 equal

- Header file: <algorithm>
- Function prototypes:
  - bool equal(InputIterator first, InputIterator last, InputIterator otherFirst);
  - bool equal(InputIterator first, InputIterator last, InputIterator otherFirst, BinaryPredicate pred);
- Description:
  - The first prototype: the elements in the range \[first, last)\] are compared to a range of equal length starting at otherFirst. The function returns true if the visited elements in both ranges are equal pairwise. The ranges need not be of equal length, only the elements in the indicated range are considered (and must be available).
  - The second prototype: the elements in the range \[first, last)\] are compared to a range of equal length starting at otherFirst. The function returns true if the binary predicate, applied to all corresponding elements in both ranges returns true for every pair of corresponding elements. The ranges need not be of equal length, only the elements in the indicated range are considered (and must be available).
- Example:

```cpp
#include <algorithm>
#include <string>
#include <cstring>
#include <iostream>
using namespace std;

class CaseString
{
public:
    bool operator()(string const &first, 
                    string const &second) const
    {
        return !strcasecmp(first.c_str(), second.c_str());
    }
};
int main()
{
    string first[] =
    {
        "Alpha", "bravo", "Charley", "delta", "Echo", 
        "foxtrot", "Golf", "hotel"
```
19.1. THE GENERIC ALGORITHMS

};
string second[] =
{
    "alpha", "bravo", "charley", "delta", "echo",
    "foxtrot", "golf", "hotel"
};
string *last = first + sizeof(first) / sizeof(string);

cout << "The elements of 'first' and 'second' are pairwise " <<
(equal(first, last, second) ? "equal" : "not equal") <<
'\n' << "compared case-insensitively, they are " <<
{equal(first, last, second, CaseString()) ? "equal" : "not equal"
} << '\n';
/*
Displays:
The elements of 'first' and 'second' are pairwise not equal
compared case-insensitively, they are equal
*/

19.1.10 equal_range

- Header file: <algorithm>

- Function prototypes:
  
  - pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first, ForwardIterator last, Type const &value);
  
  - pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first, ForwardIterator last, Type const &value, Compare comp);

- Description (see also identically named member functions, e.g., the map (section 12.4.7) and multimap (section 12.4.8)):
  
  - The first prototype: starting from a sorted sequence (where the operator< of the data type to which the iterators point was used to sort the elements in the provided range), a pair of iterators is returned representing the return value of, respectively, lower_bound (returning the first element that is not smaller than the provided reference value, see section 19.1.25) and upper_bound (returning the first element beyond the provided reference value, see section 19.1.66).
  
  - The second prototype: starting from a sorted sequence (where the comp function object was used to sort the elements in the provided range), a pair of iterators is returned representing the return values of, respectively, the functions lower_bound (section 19.1.25) and upper_bound (section 19.1.66).

- Example:

```cpp
#include <algorithm>
#include <functional>
#include <iterator>
#include <iostream>
```
using namespace std;

int main()
{
    int range[] = {1, 3, 5, 7, 7, 9, 9, 9};
    size_t const size = sizeof(range) / sizeof(int);

    pair<int *, int *> pi;
    pi = equal_range(range, range + size, 6);
    cout << "Lower bound for 6: " << *pi.first << '
';
    cout << "Upper bound for 6: " << *pi.second << '
';

    pi = equal_range(range, range + size, 7);
    cout << "Lower bound for 7: ");
    copy(pi.first, range + size, ostream_iterator<int>(cout, " ");
    cout << '
';
    cout << "Upper bound for 7: ");
    copy(pi.second, range + size, ostream_iterator<int>(cout, " ");
    cout << '
';
    sort(range, range + size, greater<int>());
    cout << "Sorted in descending order\n";
    copy(range, range + size, ostream_iterator<int>(cout, " ");
    cout << '
';
    pi = equal_range(range, range + size, 7, greater<int>());
    cout << "Lower bound for 7: ");
    copy(pi.first, range + size, ostream_iterator<int>(cout, " ");
    cout << '
';
    cout << "Upper bound for 7: ");
    copy(pi.second, range + size, ostream_iterator<int>(cout, " ");
    cout << '
';
    return 0;
}

/*
Displays:
  Lower bound for 6: 7
  Upper bound for 6: 7
  Lower bound for 7: 7 7 9 9 9
  Upper bound for 7: 9 9 9
  Sorted in descending order
  9 9 9 7 7 5 3 1
  Lower bound for 7: 7 7 5 3 1
  Upper bound for 7: 5 3 1
*/
19.1.11 **fill**

- **Header file:** `<algorithm>`

- **Function prototype:**
  ```
  void fill(ForwardIterator first, ForwardIterator last, Type const &value);
  ```

- **Description:**
  - all the elements implied by the iterator range `[first, last)` are initialized to `value`, overwriting the previously stored values.

- **Example:**
  ```
  #include <algorithm>
  #include <vector>
  #include <iterator>
  #include <iostream>
  using namespace std;

  int main()
  {
    vector<int> iv(8);
    fill(iv.begin(), iv.end(), 8);
    cout << '
';
  }
  // Displays: 8 8 8 8 8 8 8 8
  ```

19.1.12 **fill_n**

- **Header file:** `<algorithm>`

- **Function prototype:**
  ```
  void fill_n(ForwardIterator first, Size n, Type const &value);
  ```

- **Description:**
  - `n` elements starting at the element pointed to by `first` are initialized to `value`, overwriting the previously stored values.

- **Example:**
  ```
  #include <algorithm>
  #include <vector>
  #include <iterator>
  #include <iostream>
  using namespace std;

  int main()
  {
    vector<int> iv(8);
    fill_n(iv.begin(), 8, 8);
    cout << '
';
  }
  // Displays: 8 8 8 8 8 8 8 8
  ```
vector<int> iv(8);
fill_n(iv.begin() + 2, 4, 8);
copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
cout << '\n';
// Displays: 0 0 8 8 8 8 0 0

\section*{19.1.13 find}

- **Header file:** `<algorithm>`
- **Function prototype:**
  - `InputIterator find(InputIterator first, InputIterator last, Type const &value);`
- **Description:**
  - Element value is searched for in the range of the elements implied by the iterator range `[first, last)`. An iterator pointing to the first element found is returned. If the element was not found, last is returned. The operator== of the underlying data type is used to compare the elements.
- **Example:**

```cpp
#include <algorithm>
#include <string>
#include <iostream>
using namespace std;

int main()
{
    string sarr[] = {
        "alpha", "bravo", "charley", "delta", "echo"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);
    copy(
        find(sarr, last, "delta"), last,
        ostream_iterator<string>(cout, " ")
    );
    cout << '\n';
    if (find(sarr, last, "india") == last)
    {
        cout << "'india' was not found in the range\n"
        copy(sarr, last, ostream_iterator<string>(cout, " "));
        cout << '\n';
    }
}
```
19.1. THE GENERIC ALGORITHMS

/*
   Displays:
   delta echo
   'india' was not found in the range
   alpha bravo charley delta echo
*/

19.1.14 find_end

- Header file: `<algorithm`
- Function prototypes:
  - `ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2)`
  - `ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred)`
- Description:
  - The first prototype: the sequence of elements implied by `[first1, last1)` is searched for the last occurrence of the sequence of elements implied by the range `[first2, last2)`. If the sequence `[first2, last2)` is not found, `last1` is returned, otherwise an iterator pointing to the first element of the matching sequence is returned. The operator `==` of the underlying data type is used to compare the elements in the two sequences.
  - The second prototype: the sequence of elements implied by `[first1, last1)` is searched for the last occurrence of the sequence of elements implied by `[first2, last2)`. If the sequence `[first2, last2)` is not found, `last1` is returned, otherwise an iterator pointing to the first element of the matching sequence is returned. The provided binary predicate is used to compare the elements in the two sequences.
- Example:

```cpp
#include <algorithm>
#include <string>
#include <iostream>
using namespace std;

class Twice
{
   public:
      bool operator()(size_t first, size_t second) const
      {
         return first == (second << 1);
      }
};

int main()
{
   string sarr[] =
   {
      "alpha", "bravo", "charley", "delta", "echo",
```
CHAPTER 19. THE STL GENERIC ALGORITHMS

```
"foxtrot", "golf", "hotel",
"foxtrot", "golf", "hotel",
"india", "juliet", "kilo"
);
string search[] =
{
    "foxtrot",
    "golf",
    "hotel"
};
string *last = sarr + sizeof(sarr) / sizeof(string);
copy  
    (find_end(sarr, last, search, search + 3), // sequence starting  
    last, ostream_iterator<string>(cout, " ") // at 2nd 'foxtrot'  
    );
cout << '\n';
size_t range[] = {2, 4, 6, 8, 10, 4, 6, 8, 10};
size_t nrs[] = {2, 3, 4};
copy  // sequence of values starting at last sequence  
(    find_end(range, range + 9, nrs, nrs + 3, Twice()),
    range + 9, ostream_iterator<size_t>(cout, " ")
    );
cout << '\n';
/*
Displays:
foxtrot golf hotel india juliet kilo
4 6 8 10
*/

19.1.15 find_first_of

- Header file: `<algorithm`

- Function prototypes:
  - ForwardIterator1 find_first_of(ForwardIterator1 first1,  
    ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2  
    last2)
  - ForwardIterator1 find_first_of(ForwardIterator1 first1,  
    ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2  
    last2, BinaryPredicate pred)

- Description:
  - The first prototype: the sequence of elements implied by `[first1, last1)` is searched  
    for the first occurrence of an element in the sequence of elements implied by the range  
    `[first2, last2)`. If no element in the sequence `[first2, last2)` is found, `last1` is  
    returned, otherwise an iterator pointing to the first element in `[first1, last1)` that is
equal to an element in \([\text{first2}, \text{last2})\) is returned. The \texttt{operator==}\ of the underlying data type is used to compare the elements in the two sequences.

- The second prototype: the sequence of elements implied by \([\text{first1}, \text{last1})\) is searched for the first occurrence of an element in the sequence of elements implied by \([\text{first2}, \text{last2})\). Each element in the range \([\text{first1}, \text{last1})\) is compared to each element in the range \([\text{first2}, \text{last2})\), and an iterator to the first element in \([\text{first1}, \text{last1})\) for which the binary predicate \texttt{pred}\ (receiving an the element out of the range \([\text{first1}, \text{last1})\) and an element from the range \([\text{first2}, \text{last2})\)) returns \texttt{true} is returned. Otherwise, \texttt{last1} is returned.

• Example:

```cpp
#include <algorithm>
#include <string>
#include <iterator>
#include <iostream>

using namespace std;

class Twice
{
public:
    bool operator()(size_t first, size_t second) const
    {
        return first == (second << 1);
    }
};

int main()
{
    string sarr[] =
    {
        "alpha", "bravo", "charley", "delta", "echo",
        "foxtrot", "golf", "hotel",
        "foxtrot", "golf", "hotel",
        "india", "juliet", "kilo"
    };
    string search[] =
    {
        "foxtrot",
        "golf",
        "hotel"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        find_first_of(sarr, last, search, search + 3),
        last,
        ostream_iterator<string>(cout, "\n")
    );

    cout << \n';

    size_t range[] = {2, 4, 6, 8, 10, 4, 6, 8, 10};
    size_t nrs[] = {2, 3, 4};

    // copy the sequence of values in 'range', starting at the
    // first element in 'range' that is equal to twice one of the
```
// values in 'nrs', and ending at the last element of 'range'
copy
(
    find_first_of(range, range + 9, nrs, nrs + 3, Twice()),
    range + 9, ostream_iterator<size_t>(cout, " ")
);
cout << '\n';
/*
   Displays:
   foxtrot golf hotel foxtrot golf hotel india juliet kilo
   4 6 8 10 4 6 8 10
   */

19.1.16 find_if

- Header file: <algorithm>

- Function prototype:
  - InputIterator find_if(InputIterator first, InputIterator last,
                           Predicate pred);

- Description:
  - An iterator pointing to the first element in the range implied by the iterator range
    (first, last) for which the (unary) predicate pred returns true is returned. If the
    element was not found, last is returned.

- Example:

```cpp
#include <algorithm>
#include <string>
#include <cstring>
#include <iostream>
using namespace std;

class CaseName
{
    string d_string;

public:
    CaseName(char const *str): d_string(str)
    {};
    bool operator()(std::string const &element) const
    {
        return strcasecmp(element.c_str(), d_string.c_str()) == 0;
    }
};

int main()
{
    string sarr[] =
    {
        "Alpha", "Bravo", "Charley", "Delta", "Echo"
```
string *last = sarr + sizeof(sarr) / sizeof(string);

copy
{
    find_if(sarr, last, CaseName("charley")),
    last, ostream_iterator<string>(cout, " ")
};

cout << '\n';

if (find_if(sarr, last, CaseName("india")) == last)
{
    cout << "'india' was not found in the range\n";
    copy(sarr, last, ostream_iterator<string>(cout, " "));
    cout << '\n';
}
/*
Displays:
Charley Delta Echo
'india' was not found in the range
Alpha Bravo Charley Delta Echo
*/

19.1.17 for_each

- Header file: <algorithm>

- Function prototype:
  - Function for_each(ForwardIterator first, ForwardIterator last, Function func);

- Description:
  - Each of the elements implied by the iterator range [first, last) is passed in turn as a reference to the function (or function object) func. The function may modify the elements it receives (as the used iterator is a forward iterator). Alternatively, if the elements should be transformed, transform (see section 19.1.63) can be used. The function itself or a copy of the provided function object is returned: see the example below, in which an extra argument list is added to the for_each call, which argument is eventually also passed to the function given to for_each. Within for_each the return value of the function that is passed to it is ignored. The for_each generic algorithm looks a lot like the range-based for loop, but different from the range-based for-loop the for_each algorithm can also be used with sub-ranges and with reverse-iterators.

- Example:

```cpp
#include <algorithm>
#include <string>
#include <cstring>
#include <iostream>
#include <cctype>
using namespace std;
```
void lowerCase(char &c) // ‘c’ *is* modified
{
    c = tolower(static_cast<unsigned char>(c));
}

void capitalizedOutput(string const &str) // ‘str’ is *not* modified
{
    char *tmp = strcpy(new char[str.size() + 1], str.c_str());

    for_each(tmp + 1, tmp + str.size(), lowerCase);

    tmp[0] = toupper(*tmp);
    cout << tmp << " ";
    delete tmp;
}

int main()
{
    string sarr[] =
    {
        "alpha", "BRAVO", "charley", "DELTA", "echo",
        "FOXTROT", "golf", "HOTEL"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    for_each(sarr, last, capitalizedOutput)("that’s all, folks");
    cout << ‘\n’;
}

/* Displays:
   Alpha Bravo Charley Delta Echo Foxtrot Golf Hotel That’s all, folks */

• Here is another example using a function object:

#include <algorithm>
#include <string>
#include <iostream>
#include <cctype>
using namespace std;

void lowerCase(char &c)
{
    c = tolower(static_cast<unsigned char>(c));
}

class Show
{
    int d_count;

public:
    Show()
    :
        d_count(0)
    {}
    void operator() (std::string &str)
    {
        std::for_each(str.begin(), str.end(), lowerCase);
    }
}
str[0] = toupper(str[0]); // assuming str is not empty
std::cout << ++d_count << " " << str << "; ";
}
int count() const
{
    return d_count;
}
int main()
{
    string sarr[] =
    {
        "alpha", "BRAVO", "charley", "DELTA", "echo", "FOXTROT", "golf", "HOTEL"
    };
    string *last = sarr + sizeof(sarr) / sizeof(string);
    cout << for_each(sarr, last, Show()).count() << ’\n’;
    /* Displays (all on one line):
       1 Alpha; 2 Bravo; 3 Charley; 4 Delta; 5 Echo; 6 Foxtrot;
       7 Golf; 8 Hotel; 8
    */
}

The example also shows that the for_each algorithm may be used with functions defining const and non-const parameters. Also, see section 19.1.63 for differences between the for_each and transform generic algorithms.

The for_each algorithm cannot directly be used (i.e., by passing *this as the function object argument) inside a member function to modify its own object as the for_each algorithm first creates its own copy of the passed function object. A lambda function or a wrapper class whose constructor accepts a pointer or reference to the current object and possibly to one of its member functions solves this problem.

19.1.18 generate

- Header file: <algorithm>
- Function prototype:
  - void generate(ForwardIterator first, ForwardIterator last, Generator generator);
- Description:
  - All elements implied by the iterator range (first, last) are initialized by the return value of generator, which can be a function or function object. Generator::operator() does not receive any arguments. The example uses a well-known fact from algebra: in order to obtain the square of \( n + 1 \), add \( 1 + 2 \times n \) to \( n \times n \).
- Example:

  ```cpp
  #include <algorithm>
  ```
```cpp
#include <vector>
#include <iterator>
#include <iostream>
using namespace std;

class NaturalSquares
{
  size_t d_newsqr;
  size_t d_last;

  public:
    NaturalSquares(): d_newsqr(0), d_last(0) {}
    size_t operator()() // using: (a + 1)^2 == a^2 + 2*a + 1
      { return d_newsqr += (d_last++ << 1) + 1; }
};

int main()
{
  vector<size_t> uv(10);

  generate(uv.begin(), uv.end(), NaturalSquares());

  copy(uv.begin(), uv.end(), ostream_iterator<int>(cout, " "));
  cout << '\n';

  // Displays:  1 4 9 16 25 36 49 64 81 100
}
```

**19.1.19 generate_n**

- **Header file:** `<algorithm>`

- **Function prototypes:**
  - `void generate_n(ForwardIterator first, Size n, Generator generator);`

- **Description:**
  - `n` elements starting at the element pointed to by iterator `first` are initialized by the return value of `generator`, which can be a function or function object.

- **Example:**

  ```cpp
  #include <algorithm>
  #include <vector>
  #include <iterator>
  #include <iostream>
  using namespace std;

  class NaturalSquares
  {
    size_t d_newsqr;
    size_t d_last;
  };
public:
    NaturalSquares(): d_newsqr(0), d_last(0)
    {}
    size_t operator()()
    { // using: \((a + 1)^2 == a^2 + 2*a + 1\)
        return d_newsqr += (d_last++ << 1) + 1;
    }
};

int main()
{
    vector<size_t> uv(10);
    generate_n(uv.begin(), 5, NaturalSquares());
    copy(uv.begin(), uv.end(), ostream_iterator<int>(cout, " "));
    cout << 'n';
}

19.1.20 includes

- Header file: <algorithm>
- Function prototypes:
  - bool includes(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2);
  - bool includes(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, InputIterator2 last2, Compare comp);
- Description:
  - The first prototype: both sequences of elements implied by the ranges \([first1, last1)\)
    and \([first2, last2)\) should have been sorted using the operator\(< of the data type
    to which the iterators point. The function returns true if every element in the second
    sequence \([first2, last2)\) is contained in the first sequence \([first1, last1)\) (the
    second range is a subset of the first range).
  - The second prototype: both sequences of elements implied by the ranges \([first1, last1)\)
    and \([first2, last2)\) should have been sorted using the comp function object.
    The function returns true if every element in the second sequence \([first2, last2)\)
    is contained in the first sequence \([first1, last1)\) (the second range is a subset of the
    first range).
- Example:

```c++
#include <algorithm>
#include <string>
#include <cstring>
#include <iostream>
using namespace std;

class CaseString
{
    public:
```
bool operator()(string const &first,
    string const &second) const
{
    return !strcasecmp(first.c_str(), second.c_str());
}

int main()
{
    string first1[] = {
        "alpha", "bravo", "charley", "delta", "echo",
        "foxtrot", "golf", "hotel"
    };
    string first2[] = {
        "Alpha", "bravo", "Charley", "delta", "Echo",
        "foxtrot", "Golf", "hotel"
    };
    string second[] = {
        "charley", "foxtrot", "hotel"
    };
    size_t n = sizeof(first1) / sizeof(string);

    cout << "The elements of ‘second’ are " <<
        (includes(first1, first1 + n, second, second + 3) ? "" : "not")
        << " contained in the first sequence:
        "second is a subset of first1\n";

    cout << "The elements of ‘first1’ are " <<
        (includes(second, second + 3, first1, first1 + n) ? "" : "not")
        << " contained in the second sequence\n";

    cout << "The elements of ‘second’ are " <<
        (includes(first2, first2 + n, second, second + 3) ? "" : "not")
        << " contained in the first2 sequence\n";

    cout << "Using case-insensitive comparison,\n"
        "the elements of ‘second’ are " <<
        (includes(first2, first2 + n, second, second + 3, CaseString()) ? "" : "not")
        << " contained in the first2 sequence\n";
}

/*
 * Displays:
 * The elements of ‘second’ are contained in the first sequence:
 * second is a subset of first1
 * The elements of ‘first1’ are not contained in the second sequence
 * The elements of ‘second’ are not contained in the first2 sequence
 * Using case-insensitive comparison,
 * the elements of ‘second’ are contained in the first2 sequence
 */
19.1. THE GENERIC ALGORITHMS

19.1.21 inner_product

- **Header file:** `<numeric>`

- **Function prototypes:**
  
  - Type `inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, Type init);`  
  
  - Type `inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, Type init, BinaryOperator1 op1, BinaryOperator2 op2);`  

- **Description:**
  
  - The first prototype: the sum of all pairwise products of the elements implied by the range `{first1, last1}` and the same number of elements starting at the element pointed to by `first2` are added to `init`, and this sum is returned. The function uses the `operator+` and `operator*` of the data type to which the iterators point.  
  
  - The second prototype: binary operator `op1` instead of the default addition operator, and binary operator `op2` instead of the default multiplication operator are applied to all pairwise elements implied by the range `{first1, last1}` and the same number of elements starting at the element pointed to by `first2`. The results of the binary operator calls are added to `init` and `init`'s final value is returned.

- **Example:**

```cpp
#include <numeric>
#include <algorithm>
#include <iterator>
#include <iostream>

class Cat
{
    std::string d_sep;
    public:
        Cat(string const &sep)
        :
            d_sep(sep)
        {}
        string operator()(string const &s1, string const &s2) const
        {
            return s1 + d_sep + s2;
        }
};

int main()
{
    size_t ia1[] = {1, 2, 3, 4, 5, 6, 7};
    size_t ia2[] = {7, 6, 5, 4, 3, 2, 1};
    size_t init = 0;

    cout << "The sum of all squares in ";
    copy(ia1, ia1 + 7, ostream_iterator<size_t>(cout, " ") );
```
cout << "is " <<
    inner_product(ia1, ia1 + 7, ia1, init) << 'n';

cout << "The sum of all cross-products in ";
copy(ia1, ia1 + 7, ostream_iterator<size_t>(cout, " "));
cout << " and ";
copy(ia2, ia2 + 7, ostream_iterator<size_t>(cout, " "));
cout << "is " <<
    inner_product(ia1, ia1 + 7, ia2, init) << 'n';

string names1[] = {"Frank", "Karel", "Piet"};
string names2[] = {"Brokken", "Kubat", "Plomp"};

cout << "A list of all combined names in ";
copy(names1, names1 + 3, ostream_iterator<string>(cout," "));
cout << " and ";
copy(names2, names2 + 3, ostream_iterator<string>(cout," "));
cout << "is: " <<
    inner_product(names1, names1 + 3, names2, string("\t"),
        Cat("\n\t"), Cat(" ")) <<
        '\n';

/* Displays:
The sum of all squares in 1 2 3 4 5 6 7 is 140
The sum of all cross-products in 1 2 3 4 5 6 7 and 7 6 5 4 3 2 1 is 84
A list of all combined names in Frank Karel Piet and
Brokken Kubat Plomp is:
 Frank Brokken
  Karel Kubat
  Piet Plomp */

19.1.22 inplace_merge

- Header file: <algorithm>

- Function prototypes:

  - void inplace_merge(BidirectionalIterator first,
                        BidirectionalIterator middle, BidirectionalIterator last);

  - void inplace_merge(BidirectionalIterator first,
                        BidirectionalIterator middle, BidirectionalIterator last,
                        Compare comp);

- Description:

  - The first prototype: the two (sorted) ranges [first, middle) and [middle, last) are
    merged, keeping a sorted list (using the operator< of the data type to which the iterators
    point). The final series is stored in the range [first, last).

  - The second prototype: the two (sorted) ranges [first, middle) and [middle, last) are
    merged, keeping a sorted list (using the boolean result of the binary comparison operator
    comp). The final series is stored in the range [first, last).
19.1. THE GENERIC ALGORITHMS

- Example:

```cpp
#include <algorithm>
#include <string>
#include <cstring>
#include <iterator>
#include <iostream>

using namespace std;

class CaseString
{
    public:
        bool operator()(string const &first, string const &second) const
        {
            return strcasecmp(first.c_str(), second.c_str()) < 0;
        }
};

int main()
{
    // Example 1
    string range[] =
    {"alpha", "charley", "echo", "golf",
     "bravo", "delta", "foxtrot",}
    ;
    inplace_merge(range, range + 4, range + 7);
    copy(range, range + 7, ostream_iterator<string>(cout, " "));
    cout << \n;

    string range2[] =
    {"ALFA", "CHARLEY", "DELTA", "foxtrot", "hotel",
     "bravo", "ECHO", "GOLF"
    }
    ;
    inplace_merge(range2, range2 + 5, range2 + 8, CaseString());
    copy(range2, range2 + 8, ostream_iterator<string>(cout, " "));
    cout << \n;
}

/*
Displays:
alpha bravo charley delta echo foxtrot golf
ALFA bravo CHARLEY DELTA ECHO foxtrot GOLF hotel
*/

19.1.23 iter_swap

- Header file: <algorithm>

- Function prototype:

  - void iter_swap(ForwardIterator1 iter1, ForwardIterator2 iter2);

- Description:
The elements pointed to by iter1 and iter2 are swapped.

- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <string>

using namespace std;

int main()
{
    string first[] = {"alpha", "bravo", "charley"};
    string second[] = {"echo", "foxtrot", "golf"};
    size_t const n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << '\n';
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << '\n';

    for (size_t idx = 0; idx < n; ++idx)
        iter_swap(first + idx, second + idx);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << '\n';
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << '\n';
}
/*
Displays:
Before:
alpha bravo charley
echo foxtrot golf
After:
echo foxtrot golf
alpha bravo charley
*/
```

### 19.1.24 lexicographical_compare

- **Header file:** `<algorithm>`
- **Function prototypes:**
  
  - `bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);`
  
  - `bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp);`
- **Description:**
19.1. THE GENERIC ALGORITHMS

The first prototype: the corresponding pairs of elements in the ranges pointed to by the ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) are compared. The function returns true

* at the first element in the first range which is less than the corresponding element in the second range (using \(\text{operator<}\) of the underlying data type),
* if \(\text{last1}\) is reached, but \(\text{last2}\) isn't reached yet.

False is returned in the other cases, which indicates that the first sequence is not lexicographically less than the second sequence. So, \(\text{false}\) is returned:

* at the first element in the first range which is greater than the corresponding element in the second range (using \(\text{operator<}\) of the data type to which the iterators point, reversing the operands),
* if \(\text{last2}\) is reached, but \(\text{last1}\) isn't reached yet,
* if \(\text{last1}\) and \(\text{last2}\) are reached.

The second prototype: with this function the binary comparison operation as defined by \(\text{comp}\) is used instead of \(\text{operator<}\) of the data type to which the iterators point.

- Example:

```cpp
#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

class CaseString
{
public:
  bool operator()(string const &first, string const &second) const
  {
    return strcasecmp(first.c_str(), second.c_str()) < 0;
  }
};

int main()
{
  string word1 = "hello";
  string word2 = "help";

  cout << word1 << " is " << 
  (lexicographical_compare(word1.begin(), word1.end(), word2.begin(), word2.end()) ? "before "
  : "beyond or at "
  ) << 
  word2 << " in the alphabet\n";

  cout << word1 << " is " << 
  (lexicographical_compare(word1.begin(), word1.end(), word1.begin(), word1.end()) ? "before "
  : "beyond or at "
  ) << 
  word2 << " in the alphabet\n";

  return 0;
}
```

"before "
: "beyond or at "
) << 
word1 << " in the alphabet\n";
cout << word2 << " is " <<
{
 lexicographical_compare(word2.begin(), word2.end(),
 word1.begin(), word1.end()) ?
 "before "
 : "beyond or at "
} <<
word1 << " in the alphabet\n";

string one[] = {"alpha", "bravo", "charley"};
string two[] = {"ALPHA", "BRAVO", "DELTA"};
copy(one, one + 3, ostream_iterator<string>(cout, " box19.1.25 lower_bound

• Header file: <algorithm>

• Function prototypes:

  - ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const Type &value);
  - ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const Type &value, Compare comp);

• Description:
19.1. THE GENERIC ALGORITHMS

- The first prototype: the sorted elements indicated by the iterator range \([\text{first}, \text{last})\) are searched for the first element that is not less than (i.e., greater than or equal to) \(\text{value}\). The returned iterator marks the location in the sequence where \(\text{value}\) can be inserted without breaking the sorted order of the elements. The \(\text{operator}<\) of the data type to which the iterators point is used. If no such element is found, \(\text{last}\) is returned.

- The second prototype: the elements indicated by the iterator range \([\text{first}, \text{last})\) must have been sorted using the \(\text{comp}\) function (-object). Each element in the range is compared to \(\text{value}\) using the \(\text{comp}\) function. An iterator to the first element for which the binary predicate \(\text{comp}\), applied to the elements of the range and \(\text{value}\), returns \(\text{false}\) is returned. If no such element is found, \(\text{last}\) is returned.

• Example:

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <functional>
using namespace std;

int main()
{
    int ia[] = {10, 20, 30};
    cout << "Sequence: ";
    copy(ia, ia + 3, ostream_iterator<int>(cout, " "));
    cout << '
';
    cout << "15 can be inserted before " <<
         *lower_bound(ia, ia + 3, 15) << '
' ;
    cout << "35 can be inserted after " <<
         (lower_bound(ia, ia + 3, 35) == ia + 3 ?
           "the last element" : "???") << '
' ;
    iter_swap(ia, ia + 2);
    cout << "Sequence: ";
    copy(ia, ia + 3, ostream_iterator<int>(cout, " "));
    cout << '
' ;
    cout << "15 can be inserted before " <<
         *lower_bound(ia, ia + 3, 15, greater<int>()) << '
' ;
    cout << "35 can be inserted before the first element " <<
         (lower_bound(ia, ia + 3, 35, greater<int>()) == ia ?
           "the first element " : "???") << '
' ;
    return 0;
}
```

/*
Displays:
Sequence: 10 20 30
15 can be inserted before 20
35 can be inserted after the last element
Sequence: 30 20 10
15 can be inserted before 10
35 can be inserted before the first element
*/
19.1.26 max

- Header file: <algorithm>
- Function prototypes:
  - Type const &max(Type const &one, Type const &two);
  - Type const &max(Type const &one, Type const &two, Comparator comp);
- Description:
  - The first prototype: the larger of the two elements one and two is returned, using the operator> of the data type to which the iterators point to determine which element is the larger one.
  - The second prototype: one is returned if the binary predicate comp(one, two) returns true, otherwise two is returned.
- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

class CaseString
{
  public:
    bool operator()(string const &first, string const &second) const
    {
      return strcasecmp(second.c_str(), first.c_str()) > 0;
    }
};

int main()
{
  cout << "Word '" << max(string("first"), string("second")) << ", " << "is lexicographically last\n";
  cout << "Word '" << max(string("first"), string("SECOND")) << ", " << "is lexicographically last\n";
  cout << "Word '" << max(string("first"), string("SECOND"), CaseString()) << ", " << "is lexicographically last\n";
}

/*
   Displays:
   Word 'second' is lexicographically last
   Word 'first' is lexicographically last
   Word 'SECOND' is lexicographically last
*/
```

19.1.27 max_element

- Header file: <algorithm>
19.1. THE GENERIC ALGORITHMS

- **Function prototypes:**

  - ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
  - ForwardIterator max_element(ForwardIterator first, ForwardIterator last, Comparator comp);

- **Description:**

  - The first prototype: an iterator pointing to the largest element in the range implied by \([\text{first}, \text{last})\) is returned. The \texttt{operator<} of the data type to which the iterators point is used to decide which of the elements is the largest.

  - The second prototype: rather than using \texttt{operator<}, the binary predicate \texttt{comp} is used to make the comparisons between the elements implied by the iterator range \([\text{first}, \text{last})\). The element for which \texttt{comp} returns most often \texttt{true}, compared with other elements, is returned.

- **Example:**

```cpp
#include <algorithm>
#include <iostream>
using namespace std;

class AbsValue
{
    public:
    bool operator()(int first, int second) const
    {
        return abs(first) < abs(second);
    }
};

int main()
{
    int ia[] = {-4, 7, -2, 10, -12};
    cout << "The max. int value is " << *max_element(ia, ia + 5) << \n;  
    cout << "The max. absolute int value is " << 
        *max_element(ia, ia + 5, AbsValue()) << \n;
    
    /* Displays: 
       The max. int value is 10
       The max. absolute int value is -12 */
```

19.1.28 **merge**

- **Header file:** \texttt{<algorithm>}

- **Function prototypes:**

  - OutputIterator merge(InputIterator1 first1, InputIterator1 last1, 
                         InputIterator2 first2, InputIterator2 last2, OutputIterator result);
CHAPTER 19. THE STL GENERIC ALGORITHMS

OutputIterator merge(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);

• Description:

- The first prototype: the two (sorted) ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) are merged, keeping a sorted list (using the operator\(<\) of the data type to which the iterators point). The final series is stored in the range starting at \text{result} and ending just before the OutputIterator returned by the function.

- The second prototype: the two (sorted) ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) are merged, keeping a sorted list (using the boolean result of the binary comparison operator \text{comp}). The final series is stored in the range starting at \text{result} and ending just before the OutputIterator returned by the function.

• Example:

```cpp
#include <algorithm>
#include <string>
#include <cstring>
#include <iterator>
#include <iostream>
using namespace std;

class CaseString
{
public:
   bool operator()(string const &first, string const &second) const
   {
      return strcasecmp(first.c_str(), second.c_str()) < 0;
   }
};

int main()
{
   string range1[] =
   { // 5 elements
      "alpha", "bravo", "foxtrot", "hotel", "zulu"
   };
   string range2[] =
   { // 4 elements
      "delta", "echo", "golf", "romeo"
   };
   string result[5 + 4];
   copy(result,
        merge(range1, range1 + 5, range2, range2 + 4, result),
        ostream_iterator<string>(cout, " "));
   cout << \n; 
   string range3[] =
   { "ALPHA", "bravo", "foxtrot", "HOTEL", "ZULU"
   };
   string range4[] =
   { 
```
"delta", "ECHO", "GOLF", "romeo"

};

for (auto it = range3.begin(); it != range3.end(); ++it) {
    result.push_back(*it);
}

for (auto it = range4.begin(); it != range4.end(); ++it) {
    result.push_back(*it);
}

copy(result,
     merge(range3, range3 + 5, range4, range4 + 4, result,

          CaseString()),
     ostream_iterator<string>(cout, " ");

cout << '\n';
} /*
 Displays:
 alpha bravo delta echo foxtrot golf hotel romeo zulu
 ALPHA bravo delta ECHO foxtrot GOLF HOTEL romeo ZULU */

19.1.29  min

• Header file: <algorithm>

• Function prototypes:
  – Type const &min(Type const &one, Type const &two);
  – Type const &min(Type const &one, Type const &two, Comparator comp);

• Description:
  – The first prototype: the smaller of the two elements one and two is returned using the
    operator< of the data type to which the iterators point to decide which of the two ele
    ments is the smaller.
  – The second prototype: one is returned if the binary predicate comp(one, two) returns
    false, otherwise two is returned.

• Example:

#include <algorithm>
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

class CaseString
{
    public:
        bool operator() (string const &first, string const &second) const
        {
            return strcasecmp(second.c_str(), first.c_str()) > 0;
        }
};

int main()
{
    cout << "Word \" << min(string("first"), string("second")) <<
         "\" is lexicographically first\n";

    cout << "Word \" << min(string("first"), string("SECOND")) <<
        "\" is lexicographically first\n";
CHAPTER 19. THE STL GENERIC ALGORITHMS

"' is lexicographically first\n"

    cout << "Word '" << min(string("first"), string("SECOND"),
    CaseString()) << "' is lexicographically first\n";
}
/*
   Displays:
   Word 'first' is lexicographically first
   Word 'SECOND' is lexicographically first
   Word 'first' is lexicographically first
*/

19.1.30 min_element

- Header file: <algorithm>
- Function prototypes:
  - ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
  - ForwardIterator min_element(ForwardIterator first, ForwardIterator last, Comparator comp);
- Description:
  - The first prototype: an iterator pointing to the smallest element in the range implied by the range [first, last) is returned using operator< of the data type to which the iterators point to decide which of the elements is the smallest.
  - The second prototype: rather than using operator<, the binary predicate comp is used to make the comparisons between the elements implied by the iterator range [first, last). The element for which comp returns false most often is returned.
- Example:

```cpp
#include <algorithm>
#include <iostream>
using namespace std;

class AbsValue
{
    public:
        bool operator()(int first, int second) const
        {
            return abs(first) < abs(second);
        }
};

int main()
{
    int ia[] = {-4, 7, -2, 10, -12};
    cout << "The minimum int value is " << *min_element(ia, ia + 5) <<
         '\n';
    cout << "The minimum absolute int value is " <<
         *min_element(ia, ia + 5, AbsValue()) << ' \n';
```
19.1. THE GENERIC ALGORITHMS

19.1.31 mismatch

- **Header file:** `<algorithm>`
- **Function prototypes:**
  - `pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);`
  - `pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, Compare comp);`
- **Description:**
  - The first prototype: the two sequences of elements starting at `first1` and `first2` are compared using the equality operator of the data type to which the iterators point. Comparison stops if the compared elements differ (i.e., `operator==` returns false) or `last1` is reached. A pair containing iterators pointing to the final positions is returned. The second sequence may contain more elements than the first sequence. The behavior of the algorithm is undefined if the second sequence contains fewer elements than the first sequence.
  - The second prototype: the two sequences of elements starting at `first1` and `first2` are compared using the binary comparison operation as defined by `comp`, instead of `operator==`. Comparison stops if the `comp` function returns `false` or `last1` is reached. A pair containing iterators pointing to the final positions is returned. The second sequence may contain more elements than the first sequence. The behavior of the algorithm is undefined if the second sequence contains fewer elements than the first sequence.
- **Example:**

```cpp
#include <algorithm>
#include <string>
#include <cstring>
#include <iostream>
#include <utility>
using namespace std;

class CaseString
{
   public:
      bool operator()(string const &first, string const &second) const
      {
         return strcasecmp(first.c_str(), second.c_str()) == 0;
      }
};

int main()
{
   string range1[] =
```
CHAPTER 19. THE STL GENERIC ALGORITHMS

{ "alpha", "bravo", "foxtrot", "hotel", "zulu" }
string range2[] =
{ "alpha", "bravo", "foxtrot", "Hotel", "zulu" };
pair<string *, string *> pss = mismatch(range1, range1 + 5, range2);
cout << "The elements " << *pss.first << " and " << *pss.second << " at offset " << (pss.first - range1) << " differ\n";
if
    mismatch(range1, range1 + 5, range2, CaseString()).first
==
    range1 + 5
) cout << "When compared case-insensitively they match\n";
/*
Displays:
The elements hotel and Hotel at offset 3 differ
When compared case-insensitively they match
*/

19.1.32 next_permutation

- Header file: <algorithm>
- Function prototypes:
  - bool next_permutation(BidirectionalIterator first,
    BidirectionalIterator last);
  - bool next_permutation(BidirectionalIterator first,
    BidirectionalIterator last, Comp comp);
- Description:
  - The first prototype: the next permutation, given the sequence of elements in the range
    [first, last), is determined. For example, if the elements 1, 2 and 3 are the range
    for which next_permutation is called, then subsequent calls of next_permutation
    reorders the following series:
      
      1 2 3
      1 3 2
      2 1 3
      2 3 1
      3 1 2
      3 2 1

      This example shows that the elements are reordered such that each new permutation
      represents the next bigger value (132 is bigger than 123, 213 is bigger than 132, etc.) using
      operator< of the data type to which the iterators point. The value true is returned if
      a reordering took place, the value false is returned if no reordering took place, which is
      the case if the sequence represents the last (biggest) value. In that case, the sequence is
      also sorted using operator<.
The second prototype: the next permutation given the sequence of elements in the range \([\text{first}, \text{last})\) is determined, using the binary predicate \(\text{comp}\) to compare elements. The elements in the range are reordered. The value \(\text{true}\) is returned if a reordering took place, the value \(\text{false}\) is returned if no reordering took place, which is the case if the resulting sequence would haven been ordered using the binary predicate \(\text{comp}\) to compare elements.

**Example:**

```cpp
#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

class CaseString
{
    public:
    bool operator()(string const &first, string const &second) const
    {
        return strcasecmp(first.c_str(), second.c_str()) < 0;
    }
};

int main()
{
    string saints[] = {"Oh", "when", "the", "saints"};

    cout << "All permutations of 'Oh when the saints':\n";
    cout << "Sequences:\n";
    do
    {
        copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
        cout << '\n';
    } while (next_permutation(saints, saints + 4, CaseString()));

    cout << "After first sorting the sequence:\n";
    sort(saints, saints + 4, CaseString());
    cout << "Sequences:\n";
    do
    {
        copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
        cout << '\n';
    } while (next_permutation(saints, saints + 4, CaseString()));
}
/*
  Displays (partially):
  All permutations of 'Oh when the saints':
  Sequences:
  Oh when the saints
  saints Oh the when
  saints Oh when the
  saints the Oh when
*/
... After first sorting the sequence:
Sequences:
Oh saints the when
Oh saints when the
Oh the saints when
Oh the when saints
...
*/

19.1.33 nth_element

- **Header file:** `<algorithm>`
- **Function prototypes:**
  - `void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);`
  - `void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last, Compare comp);`
- **Description:**
  - The first prototype: all elements in the range `[first, last)` are sorted relative to the element pointed to by `nth`: all elements in the range `[left, nth)` are smaller than the element pointed to by `nth`, and all elements in the range `[nth + 1, last)` are greater than the element pointed to by `nth`. The two subsets themselves are not sorted. The operator `<` of the data type to which the iterators point is used to compare the elements.
  - The second prototype: all elements in the range `[first, last)` are sorted relative to the element pointed to by `nth`: all elements in the range `[left, nth)` are smaller than the element pointed to by `nth`, and all elements in the range `[nth + 1, last)` are greater than the element pointed to by `nth`. The two subsets themselves are not sorted. The `comp` function object is used to compare the elements.
- **Example:**

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <functional>
using namespace std;

int main()
{
    int ia[] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};

    nth_element(ia, ia + 3, ia + 10);
    cout << "sorting with respect to " << ia[3] << '\n';
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << '\n';

    nth_element(ia, ia + 5, ia + 10, greater<int>());
```

...
cout << "sorting with respect to " << ia[5] << '
';
copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
cout << '
';
}
/*
Displays:
  sorting with respect to 4
    1 2 3 4 9 7 5 6 8 10
  sorting with respect to 5
    10 8 7 9 6 5 3 4 2 1
*/

19.1.34 partial_sort

- Header file: <algorithm>

- Function prototypes:
  - void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last);
  - void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last, Compare comp);

- Description:
  - The first prototype: the \((\text{middle} - \text{first})\) smallest elements are sorted and stored in the range \([\text{first}, \text{middle})\) using the operator\(<\) of the data type to which the iterators point to compare elements. The remaining elements of the series remain unsorted, and are stored in the range \([\text{middle}, \text{last})\).
  - The second prototype: the \((\text{middle} - \text{first})\) smallest elements (according to the provided binary predicate \(\text{comp}\)) are sorted and stored in the range \([\text{first}, \text{middle})\). The remaining elements of the series remain unsorted.

- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int ia[] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};
    partial_sort(ia, ia + 3, ia + 10);
    cout << "find the 3 smallest elements:
";
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << "\n";
    cout << "find the 5 biggest elements:
";
    partial_sort(ia, ia + 5, ia + 10, greater<int>());
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << "\n";
    return 0;
}
```
cout << '
';
}
/*
Displays:
find the 3 smallest elements:
1 2 3 7 9 5 4 6 8 10
find the 5 biggest elements:
10 9 8 7 6 1 2 3 4 5
*/

19.1.35 partial_sort_copy

- **Header file:** `<algorithm>`
- **Function prototypes:**
  - `void partial_sort_copy(InputIterator first, InputIterator last, RandomAccessIterator dest_first, RandomAccessIterator dest_last);`
  - `void partial_sort_copy(InputIterator first, InputIterator last, RandomAccessIterator dest_first, RandomAccessIterator dest_last, Compare comp);`
- **Description:**
  - The first prototype: the \((\text{dest}_\text{last} - \text{dest}_\text{first})\) smallest elements in the range \([\text{first}, \text{last})\) are copied to the range \([\text{dest}_\text{first}, \text{dest}_\text{last})\), using the operator\(<\) of the data type to which the iterators point to decide which of the elements to copy.
  - The second prototype: the \((\text{dest}_\text{last} - \text{dest}_\text{first})\) smallest elements in the range \([\text{first}, \text{last})\) (as decided by the binary predicate \(\text{comp}\) returning \(true\)). The elements for which the predicate \(\text{comp}\) returns \(true\) most often are copied to the range \([\text{dest}_\text{first}, \text{dest}_\text{last})\).
- **Example:**

```cpp
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int ia[] = {1, 10, 3, 8, 5, 6, 7, 4, 9, 2};
    int ia2[6];

    partial_sort_copy(ia, ia + 10, ia2, ia2 + 6);

    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << '
';
    cout << "the 6 smallest elements: ";
    copy(ia2, ia2 + 6, ostream_iterator<int>(cout, " "));
    cout << '
';
```
cout << "the 4 smallest elements to a larger range:\n";
partial_sort_copy(ia, ia + 4, ia2, ia2 + 6);
copy(ia2, ia2 + 6, ostream_iterator<int>(cout, " "));
cout << '\n';

cout << "the 4 biggest elements to a larger range:\n";
partial_sort_copy(ia, ia + 4, ia2, ia2 + 6, greater<int>());
copy(ia2, ia2 + 6, ostream_iterator<int>(cout, " "));
cout << '\n';
}
/*
Displays:
1 10 3 8 5 6 7 4 9 2
the 6 smallest elements: 1 2 3 4 5 6
the 4 smallest elements to a larger range:
1 3 8 10 5 6
the 4 biggest elements to a larger range:
10 8 3 1 5 6
*/

19.1.36 partial_sum

- **Header file:** <numeric>

- **Function prototypes:**
  
  - OutputIterator partial_sum(InputIterator first, InputIterator last, OutputIterator result);
  
  - OutputIterator partial_sum(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation op);

- **Description:**
  
  - The first prototype: each element in the range [result, <returned OutputIterator>) receives a value which is obtained by adding the elements in the corresponding range of the range [first, last). The first element in the resulting range will be equal to the element pointed to by first.
  
  - The second prototype: the value of each element in the range [result, <returned OutputIterator>) is obtained by applying the binary operator op to the previous element in the resulting range and the corresponding element in the range [first, last). The first element in the resulting range will be equal to the element pointed to by first.

- **Example:**

```cpp
#include <numeric>
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int ia[] = {1, 2, 3, 4, 5};
```
int ia2[5];

copy(ia2,
    partial_sum(ia, ia + 5, ia2),
    ostream_iterator<int>(cout, " "));
cout << ‘\n’;

copy(ia2,
    partial_sum(ia, ia + 5, ia2, multiplies<int>()),
    ostream_iterator<int>(cout, " "));
cout << ‘\n’;
}
/*
Displays:
 1 3 6 10 15
 1 2 6 24 120
*/

19.1.37 partition

• Header file: <algorithm>

• Function prototype:

  - BidirectionalIterator partition(BidirectionalIterator first,
    BidirectionalIterator last, UnaryPredicate pred);

• Description:

  - All elements in the range [first, last) for which the unary predicate pred evaluates
    as true are placed before the elements which evaluate as false. The return value points
    just beyond the last element in the partitioned range for which pred evaluates as true.

• Example:

    #include <algorithm>
    #include <iostream>
    #include <string>
    #include <iterator>
    using namespace std;

    class LessThan
    {
    int d_x;
    public:
        LessThan(int x)
            :
            d_x(x)
        {}
        bool operator()(int value) const
        {
            return value <= d_x;
        }
    
    int main()
19.1. THE GENERIC ALGORITHMS

```cpp
int ia[] = {1, 3, 5, 7, 9, 10, 2, 8, 6, 4};
int *split;

split = partition(ia, ia + 10, LessThan(ia[9]));
cout << "Last element <= 4 is ia[" << split - ia - 1 << "]\n";

copy(ia, ia + 10, ostream_iterator<int>(cout, " ") );
cout << '\n';

/*
Displays:
Last element <= 4 is ia[3]
1 3 4 2 9 10 7 8 6 5
*/
```

19.1.38 prev_permutation

- **Header file**: `<algorithm>

- **Function prototypes**:
  - `bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last);`
  - `bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last, Comp comp);`

- **Description**:
  - The first prototype: the previous permutation given the sequence of elements in the range `[first, last)` is determined. The elements in the range are reordered such that the first ordering is obtained representing a ‘smaller’ value (see next_permutation (section 19.1.32) for an example involving the opposite ordering). The value `true` is returned if a reordering took place, the value `false` is returned if no reordering took place, which is the case if the provided sequence was already ordered, according to the `operator<` of the data type to which the iterators point.
  - The second prototype: the previous permutation given the sequence of elements in the range `[first, last)` is determined, using the binary predicate `comp` to compare elements. The elements in the range are reordered. The value `true` is returned if a reordering took place, the value `false` is returned if no reordering took place, which is the case if the original sequence was already ordered, using the binary predicate `comp` to compare two elements.

- **Example**:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <cstring>
#include <iterator>
using namespace std;

class CaseString {
```
public:
   bool operator()(string const &first, string const &second) const
   {
      return strcasecmp(first.c_str(), second.c_str()) < 0;
   }
};

int main()
{
   string saints[] = {"Oh", "when", "the", "saints"};

cout << "All previous permutations of 'Oh when the saints':\n";
cout << "Sequences:\n";
do
{
   copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
   cout << '\n';
} 
while (prev_permutation(saints, saints + 4, CaseString()));

cout << "After first sorting the sequence:\n";
sort(saints, saints + 4, CaseString());
cout << "Sequences:\n";
while (prev_permutation(saints, saints + 4, CaseString()))
{
   copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
   cout << '\n';
}

cout << "No (more) previous permutations\n";
} /*
 Displays:
 All previous permutations of 'Oh when the saints':
 Sequences:
 Oh when the saints
 Oh when saints the
 Oh the when saints
 Oh the saints when
 Oh saints when the
 Oh saints the when
 After first sorting the sequence:
 Sequences:
 No (more) previous permutations */

19.1.39 random_shuffle

- Header file: <algorithm>
- Function prototypes:
  - void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
  - void random_shuffle(RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator rand);
19.1. THE GENERIC ALGORITHMS

- Description:
  - The first prototype: the elements in the range \([\text{first, last})\) are randomly reordered.
  - The second prototype: The elements in the range \([\text{first, last})\) are randomly reordered using the `rand` random number generator, which should return an int in the range \([0, \text{remaining})\), where \(\text{remaining}\) is passed as argument to the `operator()` of the `rand` function object. Alternatively, the random number generator may be a function expecting an int `remaining` parameter and returning an int random value in the range \([0, \text{remaining})\). Note that when a function object is used, it cannot be an anonymous object. The function in the example uses a procedure outlined in Press et al. (1992) *Numerical Recipes in C: The Art of Scientific Computing* (New York: Cambridge University Press, (2nd ed., p. 277)).

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <time.h>
#include <iterator>
using namespace std;

int randomValue(int remaining)
{
    return rand() % remaining;
}

class RandomGenerator
{
    public:
        RandomGenerator()
        {
            srand(time(0));
        }
        int operator()(int remaining) const
        {
            return randomValue(remaining);
        }
};

void show(string *begin, string *end)
{
    copy(begin, end, ostream_iterator<string>(cout, " "));
    cout << "\n\n";
}

int main()
{
    string words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa"};
    size_t const size = sizeof(words) / sizeof(string);

    cout << "Using Default Shuffle:\n";
    random_shuffle(words, words + size);
    show(words, words + size);

    cout << "Using RandomGenerator:\n";
    RandomGenerator rg;
```
random_shuffle(words, words + size, rg);
show(words, words + size);

srand(time(0) << 1);
cout << "Using the randomValue() function:\n";
random_shuffle(words, words + size, randomValue);
show(words, words + size);
}
/*
Displays (for example):
Using Default Shuffle:
lima oscar mike november papa kilo

Using RandomGenerator:
kilo lima papa oscar mike november

Using the randomValue() function:
mike papa november kilo oscar lima
*/

19.1.40 remove

• Header file: <algorithm>

• Function prototype:
  – ForwardIterator remove(ForwardIterator first, ForwardIterator last,
    Type const &value);

• Description:
  – The elements in the range pointed to by [first, last) are reordered such that all
    values unequal to value are placed at the beginning of the range. The returned for-
    ward iterator points to the first element that can be removed after reordering. The range
    [returnvalue, last) is called the leftover of the algorithm. Note that the leftover may
    contain elements different from value, but these elements can be removed safely, as such
    elements are also present in the range [first, returnvalue). Such duplication is the
    result of the fact that the algorithm copies, rather than moves elements into new locations.
    The function uses operator== of the data type to which the iterators point to determine
    which elements to remove.

• Example:

```c++
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
  string words[] =
    { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
      "alpha", "alpha", "papa", "quebec" };
  string *removed;
```
size_t const size = sizeof(words) / sizeof(string);

cout << "Removing all \"alpha\"s:
removed = remove(words, words + size, "alpha");
copy(words, removed, ostream_iterator<string>(cout, " "));
cout << '\n'
    << "Leftover elements are:\n";
copy(removed, words + size, ostream_iterator<string>(cout, " "));
cout << '\n';
}
/*
 Displays:
 Removing all "alpha"s:
 kilo lima mike november papa quebec
 Leftover elements are:
 alpha alpha alpha papa quebec
 */

19.1.41 remove_copy

- Header file: <algorithm>
- Function prototypes:
  - OutputIterator remove_copy(InputIterator first, InputIterator last, OutputIterator result, Type const &value);
- Description:
  - The elements in the range pointed to by [first, last) not matching value are copied to the range [result, returnvalue), where returnvalue is the value returned by the function. The range [first, last) is not modified. The function uses operator== of the data type to which the iterators point to determine which elements not to copy.
- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] = {
        "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha", "oscar", "alpha", "alpha", "papa", "quebec"
    };
    size_t const size = sizeof(words) / sizeof(string);
    string remaining
    {
        size -
        count_if
        {
            words, words + size,
```
CHAPTER 19. THE STL GENERIC ALGORITHMS

bind2nd(equal_to<string>(), string("alpha"))
}

string *returnvalue =
    remove_copy(words, words + size, remaining, "alpha");

cout << "Removing all \textit{\texttt{alpha\textbackslash s}}\n";
copy(remaining, returnvalue, ostream_iterator<string>(cout, " "));
cout << '\n';
/*
Displays:
 Removing all "alpha"s:
   kilo lima mike november oscar papa quebec
*/

19.1.42 remove_copy_if

- Header file: <algorithm>
- Function prototype:
  - OutputIterator remove_copy_if(InputIterator first, InputIterator last, OutputIterator result, UnaryPredicate pred);
- Description:
  - The elements in the range pointed to by [first, last) for which the unary predicate pred returns true are removed from the resulting copy. All other elements are copied to the range [result, returnvalue), where returnvalue is the value returned by the function. The range [first, last) is not modified.
- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
    size_t const size = sizeof(words) / sizeof(string);
    string remaining[
        size -
        count_if
        (words, words + size,
         bind2nd(equal_to<string>(), "alpha")
     ];
```
19.1. THE GENERIC ALGORITHMS

string *returnvalue =
    remove_copy_if
    (words, words + size, remaining,
     bind2nd(equal_to<string>(), "alpha")
    );

cout << "Removing all \alpha\"s:
    copy(remaining, returnvalue, ostream_iterator<string>(cout, " "));
cout << '\n';
}
/*
Displays:
    Removing all "alpha"s:
        kilo lima mike november oscar papa quebec
*/

19.1.43 remove_if

- Header file: <algorithm>
- Function prototype:
  - ForwardIterator remove_if(ForwardIterator first, ForwardIterator last, UnaryPredicate pred);
- Description:
  - The elements in the range pointed to by \( \{ \text{first, last} \} \) are reordered such that all values for which the unary predicate \( \text{pred} \) evaluates as false are placed at the beginning of the range. The returned forward iterator points to the first element, after reordering, for which \( \text{pred} \) returns true. The range \( \{ \text{returnvalue, last} \} \) is called the leftover of the algorithm. The leftover may contain elements for which the predicate \( \text{pred} \) returns false, but these can safely be removed, as such elements are also present in the range \( \{ \text{first, returnvalue} \} \). Such duplication is the result of the fact that the algorithm copies, rather than moves elements into new locations.
- Example:

```cpp
#include <functional>
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
    size_t const size = sizeof(words) / sizeof(string);
    cout << "Removing all \alpha\"s:
```
string *removed = remove_if(words, words + size,
   bind2nd(equal_to<string>(), string("alpha")));

copy(words, removed, ostream_iterator<string>(cout, " "));
cout << "\n"
   << "Trailing elements are:\n";
copy(removed, words + size, ostream_iterator<string>(cout, " "));
cout << "\n";
}
/*
Displays:
Removing all "alpha"s:
kilo lima mike november oscar papa quebec
Trailing elements are:
  oscar alpha alpha papa quebec
*/

19.1.44 replace

- Header file: <algorithm>
- Function prototype:
  
  - ForwardIterator replace(ForwardIterator first, ForwardIterator last,
    Type const &oldvalue, Type const &newvalue);
- Description:
  
  - All elements equal to oldvalue in the range pointed to by [first, last) are replaced
    by a copy of newvalue. The algorithm uses operator== of the data type to which the
    iterators point.
- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
  string words[] =
    { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha", "oscar", "alpha", "alpha", "papa" };
  size_t const size = sizeof(words) / sizeof(string);

  replace(words, words + size, string("alpha"), string("ALPHA"));
  copy(words, words + size, ostream_iterator<string>(cout, " "));
  cout << "\n";
  *
  Displays
    kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA papa
  */
19.1.45 replace_copy

- Header file: `<algorithm>`

- Function prototype:
  ```cpp
  OutputIterator replace_copy(InputIterator first, InputIterator last, OutputIterator result, Type const &oldvalue, Type const &newvalue);
  ```

- Description:
  ```cpp
  All elements equal to oldvalue in the range pointed to by [first, last) are replaced by a copy of newvalue in a new range [result, returnvalue), where returnvalue is the return value of the function. The algorithm uses operator== of the data type to which the iterators point.
  ```

- Example:
  ```cpp
  #include <algorithm>
  #include <iostream>
  #include <string>
  #include <iterator>
  using namespace std;

  int main()
  {
    string words[] = {
      "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha", "oscar", "alpha", "alpha", "papa"};
    size_t const size = sizeof(words) / sizeof(string);
    string remaining[size];

    copy
    {
      remaining,
      replace_copy(words, words + size, remaining, string("alpha"),
      string("ALPHA")),
      ostream_iterator<string>(cout, " ")
   );
    cout << '\n';
  }
  /* Displays:
   kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA ALPHA papa */
  ```

19.1.46 replace_copy_if

- Header file: `<algorithm>`

- Function prototypes:
  ```cpp
  OutputIterator replace_copy_if(ForwardIterator first, ForwardIterator last, OutputIterator result, UnaryPredicate pred, Type const &value);
  ```
• Description:

- The elements in the range pointed to by \([\text{first}, \text{last})\) are copied to the range \([\text{result}, \text{returnvalue})\), where \text{returnvalue} is the value returned by the function. The elements for which the unary predicate \text{pred} returns \text{true} are replaced by \text{value}. The range \([\text{first}, \text{last})\) is not modified.

• Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha", "oscar", "alpha", "alpha", "papa"};
    size_t const size = sizeof(words) / sizeof(string);
    string result[size];

    // Note: the comparisons are: "mike" > word[i]
    replace_copy_if(words, words + size, result,
        bind1st(greater<string>(), string("mike")),
        string("ALPHA"));
    copy (result, result + size, ostream_iterator<string>(cout, " ") );
    cout << '
';
}
```

/*
   Displays (all strings in words[] which are exceeded by ‘mike’ are replaced by ALPHA):
       ALPHA ALPHA ALPHA mike ALPHA november ALPHA oscar ALPHA ALPHA papa
*/

19.1.47 replace_if

• Header file: <algorithm>

• Function prototype:

  - ForwardIterator replace_if(ForwardIterator first, ForwardIterator last, UnaryPredicate pred, Type const &value);

• Description:

  - The elements in the range pointed to by \([\text{first}, \text{last})\) for which the unary predicate \text{pred} evaluates as \text{true} are replaced by \text{value}.

Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
```
19.1. THE GENERIC ALGORITHMS

```cpp
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] = {
        "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
        "oscar", "alpha", "alpha", "papa"};
    size_t const size = sizeof(words) / sizeof(string);

    replace_if(words, words + size,
               bind1st(equal_to<string>(), string("alpha")),
               string("ALPHA"));
    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << '\n';
} /*
Displays:
kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA ALPHA papa
*/

19.1.48 reverse

- **Header file:** `<algorithm>

- **Function prototype:**

  ```cpp
  void reverse(BidirectionalIterator first, BidirectionalIterator last);
  ```

- **Description:**

  - The elements in the range pointed to by `[first, last)` are reversed.

- **Example:**

```cpp
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string line;

    while (getline(cin, line))
    {
        reverse(line.begin(), line.end());
        cout << line << '\n';
    }
}
```
19.1.49  reverse_copy

- Header file: `<algorithm`

- Function prototype:

  ```cpp
  OutputIterator reverse_copy(BidirectionalIterator first,
                             BidirectionalIterator last, OutputIterator result);
  ```

- Description:

  - The elements in the range pointed to by [first, last) are copied to the range
    [result, returnvalue) in reversed order. The value returnvalue is the value that
    is returned by the function.

- Example:

  ```cpp
  #include <algorithm>
  #include <iostream>
  #include <string>
  using namespace std;

  int main()
  {
    string line;

    while (getline(cin, line))
    {
      size_t size = line.size();
      char    copy[size + 1];

      cout << "line: " << line << '\n' << "reversed: ";
      reverse_copy(line.begin(), line.end(), copy);
      copy[size] = 0; // 0 is not part of the reversed
      // line !
      cout << copy << '\n';
    }
  }
  ```

19.1.50  rotate

- Header file: `<algorithm`

- Function prototype:

  ```cpp
  void rotate(ForwardIterator first, ForwardIterator middle,
              ForwardIterator last);
  ```

- Description:

  - The elements implied by the range [first, middle) are moved to the end of the con-
    tainer, the elements implied by the range [middle, last) are moved to the beginning
    of the container, keeping the order of the elements in the two subsets intact.
19.1. THE GENERIC ALGORITHMS

- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "lima", "mike", "november", "oscar",
          "foxtrot", "golf", "hotel", "india", "juliet" };
    size_t const size = sizeof(words) / sizeof(string);
    size_t const midsize = size / 2;
    rotate(words, words + midsize, words + size);
    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;
} /*
Displays:
   foxtrot golf hotel india juliet kilo lima mike november oscar */
```

19.1.5 rotate_copy

- Header file: <algorithm>

- Function prototypes:

  - OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last, OutputIterator result);

- Description:

  - The elements implied by the range [middle, last) and then the elements implied by [first, middle) are copied to the destination container having range [result, returnvalue), where returnvalue is the iterator returned by the function. The original order of the elements in the two subsets is not altered.

- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "lima", "mike", "november", "oscar",
          "foxtrot", "golf", "hotel", "india", "juliet" };
```
size_t const size = sizeof(words) / sizeof(string);
size_t const midsize = size / 2;
string out[size];

copy(out,
  rotate_copy(words, words + midsize, words + size, out),
  ostream_iterator<string>(cout, " "));
cout << '
';
/*
Displays:
foxtrot golf hotel india juliet kilo lima mike november oscar
*/

19.1.52 search

- Header file: <algorithm>
- Function prototypes:
  - ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);
  - ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);
- Description:
  - The first prototype: an iterator into the first range \([first1, last1)\) is returned where the elements in the range \([first2, last2)\) are found using \(\text{operator==}\) of the data type to which the iterators point. If no such location exists, \(last1\) is returned.
  - The second prototype: an iterator into the first range \([first1, last1)\) is returned where the elements in the range \([first2, last2)\) are found using the provided binary predicate \(\text{pred}\) to compare the elements in the two ranges. If no such location exists, \(last1\) is returned.
- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
using namespace std;

class absInt
{
  public:
    bool operator()(int i1, int i2) const
    {
      return abs(i1) == abs(i2);
    }
};
int main()
{
  int range1[] = {-2, -4, -6, -8, 2, 4, 6, 8};
```
int range2[] = {6, 8};

    copy
    {
        search(rangel, rangel + 8, range2, range2 + 2),
        rangel + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << 'n';

    copy
    {
        search(rangel, rangel + 8, range2, range2 + 2, absInt()),
        rangel + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << 'n';
}
/*
Displays:
  6 8
-6 -8 2 4 6 8
*/

19.1.53 search_n

- Header file: <algorithm>

- Function prototypes:

- ForwardIterator1 search_n(ForwardIterator1 first1, ForwardIterator1 last1, Size count, Type const &value);
- ForwardIterator1 search_n(ForwardIterator1 first1, ForwardIterator1 last1, Size count, Type const &value, BinaryPredicate pred);

- Description:

- The first prototype: an iterator into the first range [first1, last1) is returned where n consecutive elements having value value are found using operator== of the data type to which the iterators point to compare the elements. If no such location exists, last1 is returned.

- The second prototype: an iterator into the first range [first1, last1) is returned where n consecutive elements having value value are found using the provided binary predicate pred to compare the elements. If no such location exists, last1 is returned.

- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
using namespace std;

bool eqInt(int i1, int i2)
{
```
return abs(i1) == abs(i2);
}
int main()
{
    int range1[] = {-2, -4, -4, -6, -8, 2, 4, 4, 6, 8};

copy
(    search_n(range1, range1 + 8, 2, 4),
    range1 + 8,
    ostream_iterator<int>(cout, " ")
);
    cout << '\n';

copy
(    search_n(range1, range1 + 8, 2, 4, eqInt),
    range1 + 8,
    ostream_iterator<int>(cout, " ")
);
    cout << '\n';
}/*
Displays:
   4 4
-4 -4 -6 -8 2 4 4
*/

19.1.54 set_difference

- Header file: <algorithm>

- Function prototypes:
  - `OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
  - `OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`

- Description:
  - The first prototype: a sorted sequence of the elements pointed to by the range \([first1, last1]\) that are not present in the range \([first2, last2]\) is returned, starting at result, and ending at the OutputIterator returned by the function. The elements in the two ranges must have been sorted using operator< of the data type to which the iterators point.
  - The second prototype: a sorted sequence of the elements pointed to by the range \([first1, last1]\) that are not present in the range \([first2, last2]\) is returned, starting at result, and ending at the OutputIterator returned by the function. The elements in the two ranges must have been sorted using the comp function object.
19.1. THE GENERIC ALGORITHMS

- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <cstring>
#include <iterator>
using namespace std;

bool caseless(string const &left, string const &right)
{
    return strcasecmp(left.c_str(), right.c_str()) < 0;
}

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november", "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo"};
    string result[7];
    string *returned;

    copy(result,
         set_difference(set1, set1 + 7, set2, set2 + 3, result),
         ostream_iterator<string>(cout, " "));
    cout << '
';

    string set3[] = { "PAPA", "QUEBEC", "ROMEO"};
    copy(result,
         set_difference(set1, set1 + 7, set3, set3 + 3, result,
                        caseless),
         ostream_iterator<string>(cout, " "));
    cout << '
';
}

/*
Displays:
kilo lima mike november oscar
     kilo lima mike november oscar
*/
```

19.1.55 set_intersection

- Header file: <algorithm>

- Function prototypes:
  - OutputIterator set_intersection(InputIterator1 first1,
                                        InputIterator1 last1,
                                        InputIterator2 first2,
                                        InputIterator2 last2, OutputIterator result);
  - OutputIterator set_intersection(InputIterator1 first1,
                                        InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
                                        OutputIterator result, Compare comp);
• Description:

- The first prototype: a sorted sequence of the elements pointed to by the range \([\text{first}_1, \text{last}_1)\) that are also present in the range \([\text{first}_2, \text{last}_2)\) is returned, starting at result, and ending at the OutputIterator returned by the function. The elements in the two ranges must have been sorted using operator\(<\) of the data type to which the iterators point.

- The second prototype: a sorted sequence of the elements pointed to by the range \([\text{first}_1, \text{last}_1)\) that are also present in the range \([\text{first}_2, \text{last}_2)\) is returned, starting at result, and ending at the OutputIterator returned by the function. The elements in the two ranges must have been sorted using the \text{comp} function object.

• Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <cstring>
#include <iterator>
using namespace std;

bool caseless(string const &left, string const &right)
{
    return strcasecmp(left.c_str(), right.c_str()) < 0;
}

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november", "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo"};
    string result[7];
    string *returned;
    copy(result,
         set_intersection(set1, set1 + 7, set2, set2 + 3, result),
         ostream_iterator<string>(cout, " "));
    cout << endl;
    string set3[] = { "PAPA", "QUEBEC", "ROMEO"};
    copy(result,
         set_intersection(set1, set1 + 7, set3, set3 + 3, result, caseless),
         ostream_iterator<string>(cout, " "));
    cout << endl;
    /*
     Displays:
     papa quebec
     papa quebec
     */
```
19.1.56 set_symmetric_difference

- **Header file:** `<algorithm>`
- **Function prototypes:**
  - `OutputIterator set_symmetric_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
  - `OutputIterator set_symmetric_difference(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`
- **Description:**
  - The first prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are not present in the range `[first2, last2)` and those in the range `[first2, last2)` that are not present in the range `[first1, last1)` is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using `operator<` of the data type to which the iterators point.
  - The second prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are not present in the range `[first2, last2)` and those in the range `[first2, last2)` that are not present in the range `[first1, last1)` is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.
- **Example:**

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <cstring>
#include <iterator>
using namespace std;

bool caseless(string const &left, string const &right)
{
    return strcasecmp(left.c_str(), right.c_str()) < 0;
}

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november", "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo"};
    string result[7];

    copy(result,
         set_symmetric_difference(set1, set1 + 7, set2, set2 + 3, result),
         ostream_iterator<string>(cout, " "));
    cout << '
';

    string set3[] = { "PAPA", "QUEBEC", "ROMEO"};
```
copy(result,
    set_symmetric_difference(set1, set1 + 7, set3, set3 + 3,
                            result, caseless),
    ostream_iterator<string>(cout, " ");
cout << '
';
}
/*
 * Displays:
 * kilo lima mike november oscar romeo
 * kilo lima mike november oscar ROMEO
 */

19.1.57 set_union

- Header file: <algorithm>
- Function prototypes:
  - OutputIterator set_union(InputIterator1 first1, InputIterator1
                            last1, InputIterator2 first2, InputIterator2 last2, OutputIterator
                            result);
  - OutputIterator set_union(InputIterator1 first1, InputIterator1
                            last1, InputIterator2 first2, InputIterator2 last2, OutputIterator
                            result, Compare comp);
- Description:
  - The first prototype: a sorted sequence of the elements that are present in either the range
    [first1, last1) or the range [first2, last2) or in both ranges is returned, starting at result,
    and ending at the OutputIterator returned by the function. The elements in the two ranges must
    have been sorted using operator< of the data type to which the iterators point. Note that in the
    final range each element appears only once.
  - The second prototype: a sorted sequence of the elements that are present in either the range
    [first1, last1) or the range [first2, last2) or in both ranges is returned, starting at result,
    and ending at the OutputIterator returned by the function. The elements in the two ranges must
    have been sorted using comp function object. Note that in the final range each element appears
    only once.
- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

bool caseless(string const &left, string const &right)
{
    return strcasecmp(left.c_str(), right.c_str()) < 0;
}
int main()
{
    string set1[] = { "kilo", "lima", "mike", "november",
```
19.1. THE GENERIC ALGORITHMS

```cpp
"oscar", "papa", "quebec" );
string set2[] = { "papa", "quebec", "romeo"};
string result[8];

copy(result,
    set_union(set1, set1 + 7, set2, set2 + 3, result),
    ostream_iterator<string>(cout, " "));
cout << 'n';

string set3[] = { "PAPA", "QUEBEC", "ROMEO"};
copy(result,
    set_union(set1, set1 + 7, set3, set3 + 3, result, caseless),
    ostream_iterator<string>(cout, " "));
cout << 'n';
} /*
 Displays:
kilo lima mike november oscar papa quebec romeo
     kilo lima mike november oscar papa quebec ROMEO
 */

19.1.58 sort

- Header file: <algorithm>

- Function prototypes:
  - void sort(RandomAccessIterator first, RandomAccessIterator last);
  - void sort(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);

- Description:
  - The first prototype: the elements in the range [first, last) are sorted in ascending
    order using operator< of the data type to which the iterators point.
  - The second prototype: the elements in the range [first, last) are sorted in ascending
    order using the comp function object to compare the elements. The binary predicate comp
    should return true if its first argument should be placed earlier in the sorted sequence
    than its second argument.

- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] = {"november", "kilo", "mike", "lima",
                     "oscar", "quebec", "papa"};
```
sort(words, words + 7);
copy(words, words + 7, ostream_iterator<string>(cout, " "));
cout << '\n';

sort(words, words + 7, greater<string>());
copy(words, words + 7, ostream_iterator<string>(cout, " "));
cout << '\n';
}
/*
   Displays:
kilo lima mike november oscar papa quebec
   quebec papa oscar november mike lima kilo
*/

19.1.59 stable_partition

- Header file: <algorithm>
- Function prototype:
  - BidirectionalIterator stable_partition(BidirectionalIterator first, 
    BidirectionalIterator last, UnaryPredicate pred);
- Description:
  - All elements in the range \([\text{first}, \text{last})\) for which the unary 
    predicate \(\text{pred}\) evaluates as true are placed before the 
    elements which evaluate as false. Apart from this reordering, the 
    relative order of all elements for which the predicate evaluates to 
    false and the relative order of all elements for which the predicate 
    evaluates to true is kept. The return value points just beyond the 
    last element in the partitioned range for which \(\text{pred}\) 
    evaluates as true.
- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int org[] = {1, 3, 5, 7, 9, 10, 2, 8, 6, 4};
    int ia[10];
    int *split;

    copy(org, org + 10, ia);
    split = partition(ia, ia + 10, bind2nd(less_equal<int>(), ia[9]));
    cout << "Last element <= 4 is ia[" << split - ia - 1 << "]\n";

    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << '\n';
```
19.1. THE GENERIC ALGORITHMS

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <iterator>
using namespace std;

struct Pss: public pair<string, string> // 1
{
    Pss(string const &s1, string const &s2)
        : pair<string, string>(s1, s2) {}
};
ostream &operator<<(ostream &out, Pss const &p) // 2
{...}
```

19.1.60 stable_sort

- **Header file:** `<algorithm>`
- **Function prototypes:**
  - `void stable_sort(RandomAccessIterator first, RandomAccessIterator last);`
  - `void stable_sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`
- **Description:**
  - The first prototype: the elements in the range `[first, last)` are stable-sorted in ascending order using `operator<` of the data type to which the iterators point: the relative order of equal elements is kept.
  - The second prototype: the elements in the range `[first, last)` are stable-sorted in ascending order using the `comp` binary predicate to compare the elements. This predicate should return `true` if its first argument should be placed before its second argument in the sorted set of element.
- **Example (annotated below):**

```cpp
# include <algorithm>
# include <iostream>
# include <string>
# include <vector>
# include <iterator>
using namespace std;

struct Pss: public pair<string, string> // 1
{
    Pss(string const &s1, string const &s2)
        : pair<string, string>(s1, s2) {}
};
ostream &operator<<(ostream &out, Pss const &p) // 2
{...
```
```cpp
{ return out << " " << p.first << " " << p.second << ‘\n’;
}

class Sortby
{
    string Pss::*d_field;
    public:
        Sortby(string Pss::*field) // 3 :
            d_field(field)
        {}
        bool operator()(Pss const &p1, Pss const &p2) const // 4
        {
            return p1.*d_field < p2.*d_field;
        }
};

int main()
{
    vector<Pss> namecity; // 5
    namecity.push_back(Pss("Hampson", "Godalming"));
    namecity.push_back(Pss("Moran", "Eugene"));
    namecity.push_back(Pss("Goldberg", "Eugene"));
    namecity.push_back(Pss("Moran", "Godalming"));
    namecity.push_back(Pss("Goldberg", "Chicago"));
    namecity.push_back(Pss("Hampson", "Eugene"));

    sort(namecity.begin(), namecity.end(), Sortby(&Pss::first)); // 6
    cout << "sorted by names:
";
    copy(namecity.begin(), namecity.end(), ostream_iterator<Pss>(cout));

    stable_sort(namecity.begin(), namecity.end(), Sortby(&Pss::second));
    cout << "sorted by names within sorted cities:
";
    copy(namecity.begin(), namecity.end(), ostream_iterator<Pss>(cout));
}

/*
Displays:
sorted by names:
   Goldberg Eugene
   Goldberg Chicago
   Hampson Godalming
   Hampson Eugene
   Moran Eugene
   Moran Godalming
sorted by names within sorted cities:
   Goldberg Chicago
   Goldberg Eugene
   Hampson Eugene
   Moran Eugene
   Hampson Godalming
   Moran Godalming
*/
```
Note that the example implements a solution to an often occurring problem: how to sort using multiple hierarchal criteria. The example deserves some additional attention:

- First, at // 1 a wrapper struct Pss is created around std::pair<std::string, std::string>. Once the C++11 standard supports the direct use of base class constructors in derived classes (cf. section 13.3.3) it becomes even simpler to create this struct. The intent here is to define a type that is a wrapper around a class that is defined in the std namespace for which no insertion operation has been defined. This struct design conflicts with the principles outlined in section 14.7. However, inheritance is defensible here as the intention is not to add 'missing features' and as pair itself is in essence just Plain Old Data.

- Next (// 2), operator<< is overloaded for Pss objects. Although the compiler wouldn't have complained if this operator had been defined in the std namespace for the pair<string, string> type, this would also have been bad style as the std namespace is off limits to ordinary programs. By defining a wrapper type around pair<string, string> bad style can be prevented.

- Then (// 3), a class Sortby is defined, allowing us to construct an anonymous object receiving a pointer to one of the Pss data members that are used for sorting. In this case, as both members are string objects, its constructor can easily be defined. It expects a pointer to a string member of the class Pss.

- Sortby's operator() member (// 4) receives two references to Pss objects and uses its pointer to member to compare the appropriate fields of the Pss objects.

- In main some data is stored in a vector (// 5).

- Then (// 6) the first sort takes place. The least important criterion must be sorted first and for this a simple sort suffices. Since we want the names to be sorted within cities, the names represent the least important criterion, so we sort by names: Sortby(&Pss::first).

- The next important criterion, the cities, are sorted next (// 7). Since the relative ordering of the names are not altered anymore by stable_sort, the ties that are observed when cities are sorted are solved in such a way that the existing relative ordering is not broken. So, we end up getting Goldberg in Eugene before Hampson in Eugene, before Moran in Eugene. To sort by cities, we use another anonymous Sortby object: Sortby(&Pss::second).

19.1.61 swap

- Header file: <algorithm>

- Function prototype:
  - void swap(Type &object1, Type &object2);

- Description:
  - The elements object1 and object2 exchange their values. They do so by either cyclic copy assignment or cyclic move assignment (if available).

- Example:

```cpp
#include <algorithm>
#include <iostream>
```
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string first[] = {"alpha", "bravo", "charley"};
    string second[] = {"echo", "foxtrot", "golf"};
    size_t const n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << '\n';
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << '\n';

    for (size_t idx = 0; idx < n; ++idx)
        swap(first[idx], second[idx]);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << '\n';
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << '\n';
}

/*
Displays:
Before:
alpha bravo charley
echo foxtrot golf
After:
echo foxtrot golf
alpha bravo charley
*/

19.1.62 swap_ranges

- Header file: <algorithm>
- Function prototype:
  - ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 result);
- Description:
  - The elements in the range pointed to by [first1, last1) are swapped with the elements in the range [result, returnvalue), where returnvalue is the value returned by the function. The two ranges must be disjoint.
- Example:

#include <algorithm>
#include <iostream>
#include <string>
19.1. THE GENERIC ALGORITHMS

```c++
#include <iterator>
using namespace std;

int main()
{
    string first[] = {"alpha", "bravo", "charley"};
    string second[] = {"echo", "foxtrot", "golf"};
    size_t const n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << \n;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << \n;
    swap_ranges(first, first + n, second);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << \n;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << \n;
}
/*
Displays:
Before:
alpha bravo charley
echo foxtrot golf
After:
echo foxtrot golf
alpha bravo charley
*/
```

19.1.63 transform

- **Header file:** `<algorithm`

- **Function prototypes:**
  - `OutputIterator transform(InputIterator first, InputIterator last, OutputIterator result, UnaryOperator op);`
  - `OutputIterator transform(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, OutputIterator result, BinaryOperator op);`

- **Description:**
  - The first prototype: the unary operator op is applied to each of the elements in the range [first, last), and the resulting values are stored in the range starting at result. The return value points just beyond the last generated element.
  - The second prototype: the binary operator op is applied to each of the elements in the range [first1, last1) and the corresponding element in the second range starting at first2. The resulting values are stored in the range starting at result. The return value points just beyond the last generated element.
Example:

```cpp
#include <functional>
#include <vector>
#include <algorithm>
#include <iostream>
#include <string>
#include <cctype>
#include <iterator>
using namespace std;

string caps(string const &src)
{
    string tmp;
    tmp.resize(src.length());

    transform(src.begin(), src.end(), tmp.begin(), ::toupper);
    return tmp;
}

int main()
{
    string words[] = {"alpha", "bravo", "charley");
    copy(words, transform(words, words + 3, words, caps),
         ostream_iterator<string>(cout, " ");
    cout << '\n';

    int values[] = {1, 2, 3, 4, 5};
    vector<int> squares;

    transform(values, values + 5, values,
              back_inserter(squares), multiplies<int>());

    copy(squares.begin(), squares.end(),
         ostream_iterator<int>(cout, " ");
    cout << '\n';
}
/**
 * Displays:
 * ALPHA BRAVO CHARLEY
 * 1 4 9 16 25
 */
```

the following differences between the for_each (section 19.1.17) and transform generic algorithms should be noted:

- With transform the return value of the function object's `operator()` member is used; the argument that is passed to the `operator()` member itself is not changed.
- With for_each the function object's `operator()` receives a reference to an argument, which itself may be changed by the function object's `operator()`.

Also note that the range-based for loop can often be used instead of the transform generic algorithm. However, but different from the range-based for-loop the transform algorithm can also be used width sub-ranges and with reverse-iterators.
19.1.64 unique

- **Header file:** `<algorithm>`

- **Function prototypes:**
  
  - `ForwardIterator unique(ForwardIterator first, ForwardIterator last);`
  
  - `ForwardIterator unique(ForwardIterator first, ForwardIterator last, BinaryPredicate pred);`

- **Description:**
  
  - The first prototype: using `operator==` of the data type to which the iterators point, all but the first of consecutively equal elements in the range pointed to by `[first, last)` are relocated to the end of the range. The returned forward iterator marks the beginning of the leftover. All elements in the range `[first, return-value)` are unique, all elements in the range `[return-value, last)` are equal to elements in the range `[first, return-value)`.
  
  - The second prototype: all but the first of consecutive elements in the range pointed to by `[first, last)` for which the binary predicate `pred` (expecting two arguments of the data type to which the iterators point) returns `true`, are relocated to the end of the range. The returned forward iterator marks the beginning of the leftover. For all pairs of elements in the range `[first, return-value)` `pred` returns `false` (i.e., they are unique), while `pred` returns `true` for a combination of, as its first operand, an element in the range `[return-value, last)` and, as its second operand, an element in the range `[first, return-value)`.

- **Example:**

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <cstring>
#include <iterator>
using namespace std;

bool casestring(string const &first, string const &second)
{
    return strcasecmp(first.c_str(), second.c_str()) == 0;
}

int main()
{
    string words[] = {"alpha", "alpha", "Alpha", "papa", "quebec" };  
    size_t const size = sizeof(words) / sizeof(string);

    string *removed = unique(words, words + size);
    cout << '\n'
         << "Trailing elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " ");
    cout << '\n';

    removed = unique(words, words + size, casestring);
    copy(words, removed, ostream_iterator<string>(cout, " ");
    cout << '\n'
```
<< "Trailing elements are:\n";
copy(removed, words + size, ostream_iterator<string>(cout, " "));
cout << '\n';
} /* Displays:
   alpha Alpha papa quebec
   Trailing elements are:
   quebec
   alpha papa quebec
   Trailing elements are:
   quebec quebec */

19.1.65 unique_copy

- Header file: <algorithm>
- Function prototypes:
  - OutputIterator unique_copy(InputIterator first, InputIterator last, OutputIterator result);
  - OutputIterator unique_copy(InputIterator first, InputIterator last, OutputIterator result, BinaryPredicate pred);
- Description:
  - The first prototype: the elements in the range \([first, last)\) are copied to the resulting container, starting at \(result\). Consecutively equal elements (using operator== of the data type to which the iterators point) are copied only once (keeping the first of a series of equal elements). The returned output iterator points just beyond the last copied element.
  - The second prototype: the elements in the range \([first, last)\) are copied to the resulting container, starting at \(result\). Consecutive elements in the range pointed to by \([first, last)\) for which the binary predicate \(pred\) returns true are copied only once (keeping the first of a series of equal elements). The returned output iterator points just beyond the last copied element.
- Example:

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <iterator>
#include <cstring>
using namespace std;

bool casestring(string const &first, string const &second) {
    return strcasecmp(first.c_str(), second.c_str()) == 0;
}
int main() {
    string words[] = {"oscar", "Alpha", "alpha", "alpha",}
size_t const size = sizeof(words) / sizeof(string);
vector<string> remaining;
unique_copy(words, words + size, back_inserter(remaining));

copy(remaining.begin(), remaining.end(),
    ostream_iterator<string>(cout, " "));
cout << '
';

vector<string> remaining2;
unique_copy(words, words + size,
    back_inserter(remaining2), casestring);

copy(remaining2.begin(), remaining2.end(),
    ostream_iterator<string>(cout, " "));
cout << '
';
} /*
 * Displays:
 * oscar Alpha alpha papa quebec
 * oscar Alpha papa quebec
 */

19.1.66 upper_bound

- **Header file:** `<algorithm>`

- **Function prototypes:**
  - `ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, Type const &value);`
  - `ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, Type const &value, Compare comp);`

- **Description:**
  - The first prototype: the sorted elements (using ascending sort) stored in the iterator range `(first, last)` are searched for the first element that is greater than `value`. The returned iterator marks the first location in the sequence where `value` can be inserted without breaking the sorted order of the elements using `operator<` of the data type to which the iterators point. If no such element is found, `last` is returned.
  - The second prototype: the elements implied by the iterator range `[first, last)` must have been sorted using the `comp` function or function object. Each element in the range is compared to `value` using the `comp` function. An iterator to the first element for which the binary predicate `comp`, applied to the elements of the range and `value`, returns `true` is returned. If no such element is found, `last` is returned.

- **Example:**

```cpp
#include <algorithm>
#include <iostream>
#include <functional>
```
#include <iterator>
using namespace std;

int main()
{
    int ia[] = {10, 15, 15, 20, 30};
    size_t n = sizeof(ia) / sizeof(int);

    cout << "Sequence: ";
    copy(ia, ia + n, ostream_iterator<int>(cout, " ") );
    cout << '
';
    cout << "15 can be inserted before " << *upper_bound(ia, ia + n, 15) << '\n';
    cout << "35 can be inserted after " <<
    (upper_bound(ia, ia + n, 35) == ia + n ?
    "the last element" : "???") << '\n';

    sort(ia, ia + n, greater<int>());

    cout << "Sequence: ";
    copy(ia, ia + n, ostream_iterator<int>(cout, " ") );
    cout << '
';
    cout << "15 can be inserted before " <<
    *upper_bound(ia, ia + n, 15, greater<int>()) << '\n';
    cout << "35 can be inserted before " <<
    (upper_bound(ia, ia + n, 35, greater<int>()) == ia ?
    "the first element" : "???") << '\n';
}

/*
Displays:
Sequence: 10 15 15 20 30
15 can be inserted before 20
35 can be inserted after the last element
Sequence: 30 20 15 15 10
15 can be inserted before 10
35 can be inserted before the first element
*/

19.1.67 Heap algorithms

A heap is a kind of binary tree which can be represented by an array. In the standard heap, the key of an element is not smaller than the key of its children. This kind of heap is called a max heap. A tree in which numbers are keys could be organized as shown in figure 19.1. Such a tree may also be organized in an array:

12, 11, 10, 8, 9, 7, 6, 1, 2, 4, 3, 5

In the following description, keep two pointers into this array in mind: a pointer node indicates the location of the next node of the tree, a pointer child points to the next element which is a child of the node pointer. Initially, node points to the first element, and child points to the second element.
19.1. THE GENERIC ALGORITHMS

Figure 19.1: A binary tree representation of a heap

- \*node++ (== 12). 12 is the top node. its children are \*child++ (11) and \*child++ (10), both less than 12.
- The next node (\*node++ (== 11)), in turn, has \*child++ (8) and \*child++ (9) as its children.
- The next node (\*node++ (== 10)) has \*child++ (7) and \*child++ (6) as its children.
- The next node (\*node++ (== 8)) has \*child++ (1) and \*child++ (2) as its children.
- Then, node (\*node++ (== 9)) has children \*child++ (4) and \*child++ (3).
- Finally (as far as children are concerned) (\*node++ (== 7)) has one child \*child++ (5)

Since child now points beyond the array, the remaining nodes have no children. So, nodes 6, 1, 2, 4, 3 and 5 don’t have children.

Note that the left and right branches are not ordered: 8 is less than 9, but 7 is larger than 6.

A heap is created by traversing a binary tree level-wise, starting from the top node. The top node is 12, at the zeroth level. At the first level we find 11 and 10. At the second level 8, 9, 7 and 6 are found, etc.

Heaps can be constructed in containers supporting random access. So, a list is not an appropriate data structure for a heap. Heaps can be constructed from an (unsorted) array (using make_heap). The top-element can be pruned from a heap, followed by reordering the heap (using pop_heap), a new element can be added to the heap, followed by reordering the heap (using push_heap), and the elements in a heap can be sorted (using sort_heap, which, of course, invalidates the heap).

19.1.67.1 The ‘make_heap’ function

- Header file: <algorithm>
- Function prototypes:
  - void make_heap(RandomAccessIterator first, RandomAccessIterator last);
  - void make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
CHAPTER 19. THE STL GENERIC ALGORITHMS

- Description:
  - The first prototype: the elements in the range \([\text{first}, \text{last})\) are reordered to form a max-heap using \(\text{operator}<\) of the data type to which the iterators point.
  - The second prototype: the elements in the range \([\text{first}, \text{last})\) are reordered to form a max-heap using the binary comparison function object \(\text{comp}\) to compare elements.

19.1.67.2 The ‘pop_heap’ function

- Header file: <algorithm>
- Function prototypes:
  - void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
  - void pop_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
- Description:
  - The first prototype: the first element in the range \([\text{first}, \text{last})\) is moved to \(\text{last} - 1\). Then, the elements in the range \([\text{first}, \text{last} - 1)\) are reordered to form a max-heap using the \(\text{operator}<\) of the data type to which the iterators point.
  - The second prototype: the first element in the range \([\text{first}, \text{last})\) is moved to \(\text{last} - 1\). Then, the elements in the range \([\text{first}, \text{last} - 1)\) are reordered to form a max-heap using the binary comparison function object \(\text{comp}\) to compare elements.

19.1.67.3 The ‘push_heap’ function

- Header file: <algorithm>
- Function prototypes:
  - void push_heap(RandomAccessIterator first, RandomAccessIterator last);
  - void push_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
- Description:
  - The first prototype: assuming that the range \([\text{first}, \text{last} - 1)\) contains a valid heap, and the element at \(\text{last} - 1\) contains an element to be added to the heap, the elements in the range \([\text{first}, \text{last} - 1)\) are reordered to form a max-heap using the \(\text{operator}<\) of the data type to which the iterators point.
  - The second prototype: assuming that the range \([\text{first}, \text{last} - 1)\) contains a valid heap, and the element at \(\text{last} - 1\) contains an element to be added to the heap, the elements in the range \([\text{first}, \text{last} - 1)\) are reordered to form a max-heap using the binary comparison function object \(\text{comp}\) to compare elements.
19.1. THE GENERIC ALGORITHMS

19.1.67.4 The 'sort_heap' function

- **Header file:** `<algorithm>

- **Function prototypes:**
  - `void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
  - `void sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

- **Description:**
  - The first prototype: assuming the elements in the range `[first, last)` form a valid max-heap, the elements in the range `[first, last)` are sorted using `operator<` of the data type to which the iterators point.
  - The second prototype: assuming the elements in the range `[first, last)` form a valid heap, the elements in the range `[first, last)` are sorted using the binary comparison function object `comp` to compare elements.

19.1.67.5 An example using the heap functions

Here is an example showing the various generic algorithms manipulating heaps:

```cpp
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

void show(int *ia, char const *header)
{
    cout << header << ":\n";
    copy(ia, ia + 20, ostream_iterator<int>(cout, " "));
    cout << '\n';
}

int main()
{
    int ia[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20};

    make_heap(ia, ia + 20);
    show(ia, "The values 1-20 in a max-heap");

    pop_heap(ia, ia + 20);
    show(ia, "Removing the first element (now at the end)");

    push_heap(ia, ia + 20);
    show(ia, "Adding 20 (at the end) to the heap again");

    sort_heap(ia, ia + 20);
    show(ia, "Sorting the elements in the heap");
```


make_heap(ia, ia + 20, greater<int>());
show(ia, "The values 1-20 in a heap, using > (and beyond too)");  

pop_heap(ia, ia + 20, greater<int>());
show(ia, "Removing the first element (now at the end)");  

push_heap(ia, ia + 20, greater<int>());
show(ia, "Re-adding the removed element");  

sort_heap(ia, ia + 20, greater<int>());
show(ia, "Sorting the elements in the heap");

} /* Displays:
The values 1-20 in a max-heap:
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Removing the first element (now at the end):
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Adding 20 (at the end) to the heap again:
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Sorting the elements in the heap:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
The values 1-20 in a heap, using > (and beyond too):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Removing the first element (now at the end):
2 4 3 8 5 6 7 16 9 10 11 12 13 14 15 20 17 18 19 1
Re-adding the removed element:
1 2 3 4 5 6 7 16 9 5 11 12 13 14 15 20 17 18 19 10
Sorting the elements in the heap:
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 */

19.2 STL: More function adaptors

Before using the function adaptors presented in these (sub)sections the <functional> header file must be included.

Member function adaptors are part of the Standard Template Library (STL). They are most useful in combination with the generic algorithms, which is why they are discussed in this chapter instead of the previous chapter which was devoted to the STL.

The member function adaptors defined in the STL allow us to call (parameterless) member functions of class objects as though they were non-member functions and to compose unary or binary argument functions into single function objects so that they can jointly be used with generic algorithms.

In the next section calling member functions as non-member functions is discussed; adaptable functions are covered thereafter.
19.2. STL: MORE FUNCTION ADAPTORS

19.2.1 Member function adaptors

Member function adaptors allow the use of generic algorithms when the function to call is a member function of a class. Consider the following class:

```cpp
class Data
{
  public:
    void display();
};
```

Obviously, the `display` member is used to display the information stored in `Data` objects.

When storing `Data` objects in a vector the `for_each` generic algorithm cannot easily be used to call the `display` member for each of the objects stored in the vector, as `for_each` accepts either a (non-member) function or a function object as its third argument, but does not accept a member function.

The member function adaptor `mem_fun_ref` can be used to solve this problem. It expects the address of a member function without any parameters and its function call operator calls that function for the object that is passed to its function call operator. In the next example `vector<Data> data` is filled with `Data` objects. Then `for_each` is used to display the information in the various `Data` objects that are stored in the `data` vector:

```cpp
int main()
{
  vector<Data> data;
  // the 'data' vector is filled with Data objects here
  for_each(data.begin(), data.end(), mem_fun_ref(&Data::display));
}
```

A second member function adaptor is `mem_fun`, which is used to call a member function from a pointer to an object. The above example could be rewritten to something like:

```cpp
int main()
{
  vector<Data *> data;
  data.push_back(new Data); // multiple push-backs if applicable
  for_each(data.begin(), data.end(), mem_fun(&Data::display));
  // delete the Data objects pointed to by data’s elements
}
```

An interesting observation is that [http://www.sgi.com](http://www.sgi.com) provides an example of the use of `mem_fun` where polymorphic members are used. If `Data::display` is a virtual member function and `Derived1` and `Derived2` (both derived from `Data`) provide their own implementations of `display`, then pointers to `Derived1` and `Derived2` objects can be entered into the `data` vector. The `for_each` algorithm then calls the proper (overridden) `display` function. E.g.,

```cpp
int main()
```
\[
\begin{align*}
\{ & \\
\text{vector<Data *> data;} & \\
\text{data.push_back(new Derived1);} & \\
\text{data.push_back(new Derived2);} & \\
\text{// calls Derived1::display or} & \\
\text{// Derived2::display.} & \\
\text{for_each(data.begin(), data.end(), mem_fun(&Data::display));} & 
\end{align*}
\]

This example, however, uses public virtual members, confounding the virtual and public interfaces of classes (cf. section 14.7) and is therefore deprecated. Polymorphism could still be used, though, but the public interface should be provided by Data as follows:

```cpp
class Data
{
    public:
        void display(); // calls v_display

    private:
        virtual void v_display(); // overridden by derived
}; // classes
```

In the above example `mem_fun` would simply receive `&Data::display`'s address. No need to modify the above `main` function, but now Data's polymorphic features are completely separated from its public interface.

### 19.2.2 Adaptable functions

Within the context of the STL an adaptable function is a function object defining

- argument_type as a synonym of the type of its unary function operator's argument, and
- first_argument_type and second_argument_type as synonyms of the types of its binary function operator's arguments

and defining result_type as the type of the return value of its function call operator.

The STL defines `pointer_to_unary_function` and `pointer_to_binary_function` as adaptors accepting pointers to, respectively, unary and binary functions, converting them into adaptable functions. In the context of the STL using adaptable functions should probably be preferred over pointers to functions as the STL's adaptable functions define the abovementioned types describing their arguments and result types. At the end of this section an example is given where these types are actually required, and a plain pointer to a function cannot be used.

In practice the `pointer_to_unary_function` and `pointer_to_binary_function` adaptors are hardly ever explicitly used. Instead, the overloaded function adaptor `ptr_fun` is used.

When `ptr_fun` is provided with a unary function it uses `pointer_to_unary_function` to create an adaptable unary function and it uses `pointer_to_binary_function` when provided with a binary function to create an adaptable binary function.

Here is an example showing the use of `ptr_fun`, creating an adaptable binary function. In `main`, the word to search for is extracted from the standard input stream as well as additional words that
are stored in a vector of `string` objects. If the target word is found the program displays the word following the target word in the vector of `string` objects. Searching is performed case insensitively, for which the POSIX function `strcasicmp` is used:

```cpp
#include <vector>
#include <algorithm>
#include <functional>
#include <cstring>
#include <string>
#include <iterator>
#include <iostream>

using namespace std;

inline int stringcasecmp(string lhs, string rhs)
{
    return strcasecmp(lhs.c_str(), rhs.c_str());
}

int main()
{
    string target;
    cin >> target;

    vector<string> v1;
    copy(istream_iterator<string>(cin), istream_iterator<string>(),
        back_inserter(v1));

    auto pos = find_if(
        v1.begin(), v1.end(),
        not1( bind2nd(ptr_fun(stringcasecmp), target) )
    );

    if ( pos != v1.end())
        cout << "The search for """ << target << "' was successful.\n"
        "The next string is: """ << pos[1] << "'.\n";
}
```

The observant reader may have noticed that this is not a very efficient program. The function `stringcasecmp` defines value type parameters forcing the construction of copies of the arguments every time it is called. However, when changing the parameter definitions into

```cpp
inline int stringcasecmp(string const &lhs, string const &rhs)
```

the compiler generates an error message like:

```cpp
In instantiation of ‘std::binder2nd<std::pointer_to_binary_function<
```
const std::string&, const std::string&, int> >':

typename _Operation::result_type std::binder2nd<_Operation>::operator()(typename _Operation::first_argument_type&) const
cannot be overloaded with:

typename _Operation::result_type std::binder2nd<_Operation>::operator()(const typename _Operation::first_argument_type&) const

This problem (and its solution) is covered in a later chapter (cf. section 22.5.2.1).
Chapter 20

Multi Threading

The 98 C++ standard did not acknowledge the existence of multi-threading. Between then and the release of the C++11 standard computers have evolved to multi-core machines, and using multi-threading by now is a real option to consider when developing software.

Multi-threading is an extensive and complex subject, and many good reference texts on the subject exist. The C++ multi-threading is built upon the facilities offered by the pthreads library (cf. Nichols, B, et al.’s Pthreads Programming\(^1\), O’Reilly). However, in line with C++’s current-day philosophy the multi-threading implementation offered by the C++11 standard offers a high level interface to multi-threading, and using the raw pthread building blocks is hardly ever necessary (cf. Williams, A. (2012): C++ Concurrency in action, Manning).

In this chapter the facilities for multi-threading as offered by the C++11 standard are covered. Although the coverage aims at providing the tools and examples allowing you to create your own multi-threaded programs, coverage necessarily is far from complete. The topic of multi-threading is too extensive for that. The mentioned reference texts provide a good starting point for any further study of multi-threading.

A thread of execution (commonly abbreviated to a thread) is a single flow of control within a program. It differs from a separately executed program, as created by the fork(1) system call in the sense that threads all run inside one program, while fork(1) creates independent copies of a running program. Multi-threading means that multiple tasks are being executed in parallel inside one program, and no assumptions can be made as to which thread is running first or last, or at what moment in time. Especially when the number of threads does not exceed the number of cores, each thread may be active at the same time. If the number of threads exceed the number of cores, the operating system will resort to task switching, offering each thread time slices in which it can perform its tasks. Task switching takes time, and the law of diminishing returns applies here as well: if the number of threads greatly exceeds the number of available cores (also called overpopulation), then the overhead incurred may exceed the benefit of being able to run multiple tasks in parallel.

Since all threads are running inside one single program, all threads share the program’s data and code. When the same data are accessed by multiple threads, and at least one of the threads is modifying these data, access must be synchronized to avoid that threads read data while these data are being modified by other threads, and to avoid that multiple threads modify the same data at the same time.

So how do we run a multi-threaded program in C++? Let’s look at hello world, the multi-threaded way:

\(^1\)http://oreilly.com/catalog/
1: #include <iostream>
2: #include <thread>
3: void hello()
4: {
5:   std::cout << "hello world!\n";
6: }
7: }
8: int main()
9: {
10:   std::thread hi(hello);
11:   hi.join();
12:   }

• At line 2 the header thread is included, informing the compiler about the existence of the class std::thread (cf. section 20.2.2);

• At line 11 the std::thread hi object is created. It is provided with the name of a function (hello) which will be called in a separate thread. Actually, the second thread, running hello, is immediately started when a std::thread is defined this way;

• The main function itself also represents a thread: the program’s first thread. It should wait until the second thread has finished. This is realized in line 12, where hi.join() waits until the thread hi has finished its job. Since there are no further statements in main, the program itself ends immediately thereafter:

• The function hello itself, defined in lines 4 through 7, is trivial: it simply inserts the text ‘hello world’ into cout, and terminates, thus ending the second thread.

When compiling multi-threaded programs using the Gnu g++ compiler the -pthread option must be specified. At link-time the libpthread library must be available as well.

To create a multi-threaded program defined in a source file multi.cc the g++ compiler can be called like this:

```
g++ --std=c++11 -pthread -Wall multi.cc
```

When several pre-compiled objects must be linked, the -lpthread linker option should also be specified.

### 20.1 Handling time (absolute and relative)

The C programming language offered tools like sleep(3) and select(2) to suspend program execution for a certain amount of time.

In multi threaded programs threads are frequently suspended, albeit usually for a very short time interval. E.g., when a thread must access a variable, but the variable is currently updated by another thread, then the first thread should wait until the second thread has completed the update. Updating a variable usually doesn’t take much time, but if it may take an unexpectedly long time, then the waiting thread may want to be informed about it, so it can do something else as long as the second thread hasn’t finished updating the variable.
20.1. HANDLING TIME (ABSOLUTE AND RELATIVE)

Sleep and select can be used for waiting, but as they were designed in an era when multithreading was commonly unavailable, their capabilities are limited when used in multithreaded programs.

The gap is bridged by the STL offering dedicated classes for specifying time, which mix well with time-dependent thread members. Threads are the topic of the next section (20.2). Before that, we’ll first have a look at facilities for specifying time.

### 20.1.1 Units: the class std::ratio

Thread execution may have to be suspended until a specific point in time, for a specific amount of time, or until some event has occurred. When specifying time an appropriate time unit must be selected. Units (not necessarily time units) are defined in the class template std::ratio.

Before the class ratio can be used, the <ratio> header file must be included. Usually just the <chrono> header file is included, as chrono includes ratio, and facilities for specifying time are available after including the chrono header file.

The class template ratio expects two integral template arguments, defining, respectively, the numerator (amount) and denominator (fraction) of an amount. By default the denominator equals 1, resulting in the ratio’s first argument (the numerator) being interpreted as the represented amount. Examples:

```cpp
ratio<1> - representing one;
ratio<60> - representing 60
ratio<1, 1000> - representing 1/1000.
```

The class template ratio defines two directly accessible static data fields: num represents its numerator, den its denominator. A ratio definition by itself simply defines a certain amount. E.g., when executing the following program

```cpp
#include <ratio>
#include <iostream>
using namespace std;
int main()
{
    cout << ratio<5, 1000>::num << ',' << ratio<5, 1000>::den << '\n';
}
```

the text 1,200 is displayed, as that’s the ‘amount’ represented by ratio<5, 1000>::ratio simplifies the fraction whenever possible.

A fairly large number of predefined ratio types exist. They can be used instead of the more cumbersome ratio<x> or ratio<x, y> specification:

<table>
<thead>
<tr>
<th>yocto</th>
<th>zepto</th>
<th>atto</th>
<th>femto</th>
<th>pico</th>
<th>nano</th>
<th>micro</th>
<th>milli</th>
<th>milli</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^-24</td>
<td>10^-21</td>
<td>10^-18</td>
<td>10^-15</td>
<td>10^-12</td>
<td>10^-9</td>
<td>10^-6</td>
<td>10^-3</td>
<td>10^-3</td>
</tr>
<tr>
<td>centi</td>
<td>deci</td>
<td>hecto</td>
<td>kilo</td>
<td>10^3</td>
<td>mega</td>
<td>giga</td>
<td>tera</td>
<td>10^12</td>
</tr>
<tr>
<td>10^-2</td>
<td>10^1</td>
<td>10^4</td>
<td>10^9</td>
<td>10^12</td>
<td>10^6</td>
<td>10^9</td>
<td>10^15</td>
<td>10^18</td>
</tr>
<tr>
<td>peta</td>
<td>10^15</td>
<td>zetta</td>
<td>10^21</td>
<td>10^24</td>
<td>yotta</td>
<td>10^24</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(note: the definitions of the types yocto, zepto, zetta and yotta use integral constants exceeding 64 bits. Although these constants are defined in the C++11 standard, they are not available on 64 bit or smaller architectures.)

20.1.2 Amounts of time: std::chrono::duration

The class template std::chrono::duration is defined in the std::chrono namespace. Objects of the class duration define amounts of time.

Before using the class duration the <chrono> header file must be included (which in turn includes the <ratio> header file).

The class template duration requires two template type arguments: a numeric type (commonly int64_t) defining the duration's value, and a time-unit, called its Period, usually defined using the class template ratio.

Here predefined ratio types simplify the job of using the right granularity. E.g., to define 30 minutes you could use

```cpp
std::chrono::duration<int64_t, std::deca> halfHr(180)
```

(resulting in 180 deca-seconds, so 1800 seconds) but even if you specify 'using namespace std' and 'using namespace chrono' this is rather complex and non-intuitive. Fortunately, various predefined duration types exist:

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>nanoseconds</td>
<td>duration&lt;int64_t, nano&gt;</td>
</tr>
<tr>
<td>microseconds</td>
<td>duration&lt;int64_t, micro&gt;</td>
</tr>
<tr>
<td>milliseconds</td>
<td>duration&lt;int64_t, milli&gt;</td>
</tr>
<tr>
<td>seconds</td>
<td>duration&lt;int64_t&gt;</td>
</tr>
<tr>
<td>minutes</td>
<td>duration&lt;int64_t, ratio&lt;60&gt;&gt;</td>
</tr>
<tr>
<td>hours</td>
<td>duration&lt;int64_t, ratio&lt;3600&gt;&gt;</td>
</tr>
</tbody>
</table>

Using these types, a time amount of 30 minutes can now simply be defined as

```cpp
std::chrono::minutes halfHour(30).
```

The class template duration itself defines two types:

- std::chrono::duration<Value, Period>::rep: the Value type used by the duration type (e.g., int64_t)
- std::chrono::duration<Value, Period>::period: the ratio type used by the duration (e.g., std::ratio::nano) (so period itself has static values num and (den), as mentioned in the previous section).

These types can be retrieved from a duration object using decltype. E.g.,

```cpp
auto time(minutes(3) * 3);
cout << decltype(time)::period::num; // displays 60
```

In addition to these types the class template duration offers the following constructors:

- constexpr duration(): the default constructor defines a duration of zero time units.
• constexpr explicit duration(Value const &value): a specific duration of value time units. Here Value refers to the duration’s numeric type (e.g., int64_t). So, when defining

```cpp
std::chrono::minutes halfHour(30);
```

the argument 30 is stored inside an int64_t.

• duration also offers copy and move constructors.

In addition, duration has these members:

• Arithmetic operators:

  Duration objects may be added, or subtracted, and they may be multiplied and divided by a numeric value. They also support the modulo operator, using an integral constant for its right-hand side operand. The binary arithmetic and the binary compound assignment operators are also available.

• constexpr Value count() const:

  returns the value that is stored inside a duration object. For halfHour this would return 30, not 1800, as the time unit itself is obtained from its duration<Value, Unit>::period type.

• static constexpr duration zero():

  this is a static member, returning a duration object representing a duration of zero time units.

• static constexpr duration min():

  this is a static member, returning a duration object representing numeric_limits<Rep>::lowest().

• static constexpr duration max():

  this is a static member, returning a duration object representing numeric_limits<Rep>::max().

Different duration types may be combined, unless precision would be lost. When the binary arithmetic operators are used the resulting duration uses the finer of the two granularities. When the binary compound assignment operator is used the granularity of the left-hand side operand must at least be equal to the granularity of the right-hand side operand, or a compilation error is issued. E.g.,

```cpp
minutes halfHour(30);
seconds half_a_minute(30);

cout << (halfHour + half_a_minute).count(); // displays 1830

// halfHour += half_a_minute; won’t compile: precision is lost

half_a_minute += halfHour;
cout << half_a_minute.count(); // displays 1830
```
20.1.3 Clocks measuring time

Since time plays an important role, not only in real life, but also in multi-threaded programs, we need clocks to represent and measure time. The C++11 standard offers several predefined clock types, and all these clocks are defined in the `std::chrono` namespace.

Before using these clocks the `<chrono>` header file must be included.

A clock type must be specified when referring to a point in time using `std::chrono::time_point` (covered by the next section). It is also possible to define your own clock type (not covered by the C++ Annotations (clause 20.11.3 of the C++11 standard lists the requirements for a clock type)).

If `Clock` is a predefined clock type, then `Clock` defines the following types:

- `std::chrono::Clock::duration`:
  the duration type which used by `Clock` (e.g., `std::chrono::nanoseconds`)
- `std::chrono::Clock::period`:
  the time period used by `Clock` (e.g., `std::ratio::nano`)
- `std::chrono::Clock::rep`:
  defining the value type used by `Clock` to store duration values (e.g., `int64_t`)
- `std::chrono::Clock::time_point`:
  time points used by `Clock` (e.g., `std::chrono::time_point<system_clock, duration>`)  

In addition to these types predefined clocks offer a member

- `static std::chrono::time_point std::chrono::Clock::now()`:  
  returns the `time_point` representing the current time

There are three predefined clock types:

- `std::chrono::system_clock`:
  the ‘wall clock’ time, using the system’s real time clock. In addition to `now` the class `system_clock` offers these two static members:
  - `static time_t std::chrono::system_clock::to_time_t (std::chrono::time_point const & timePoint)`  
    a `time_t` value (the same type as returned by C’s `time(2)` function) as used by the representing the same point in time as `timePoint`. Example:
    ```cpp
    std::chrono::system_clock::to_time_t (std::chrono::system_clock().now())
    ```  
  - `static std::chrono::time_point std::chrono::system_clock::from_time_t (time_t seconds)`  
    a `time_point` value representing the same point in time as `time_t`.
- `std::chrono::steady_clock`:
  a clock whose time increases in parallel with the increase of real time.
20.1. HANDLING TIME (ABSOLUTE AND RELATIVE)

- std::chrono::high_resolution_clock:
  the computer’s fastest clock (i.e., the clock having the shortest timer-tick period). In practice this is the same clock as system_clock.

As an example: to access the current time you could use:

```cpp
auto point = std::chrono::system_clock::now();
```

20.1.4 Points in time: std::chrono::time_point

The class time_point is defined in the std::chrono namespace. Objects of the class std::chrono::time_point define a point in time.

Before using the class std::chrono::time_point the <chrono> header file must be included.

The class time_point is a class template, requiring two template type arguments: a Clock type and a Duration type. The Clock type usually is one of the predefined clock types, e.g., std::chrono::system_clock. The Duration type may be omitted, in which case the Clock’s duration type is used. An explicit duration type may also be provided.

In the previous section auto was used to specify the type of the return value of std::chrono::system_clock::now. The explicit definition looks like this:

```cpp
std::chrono::time_point<std::chrono::system_clock> now = std::chrono::system_clock::now();
```

The class std::chrono::time_point features three constructors:

- time_point():
  the default constructor represents the beginning of the clock’s epoch (E.g., Jan, 1, 1970, 00:00h);

- time_point(time_point<Clock, Duration> const &timeStep):
  initializes a time_point object to represent a point in time timeStep Duration units beyond the clock’s epoch;

- time_point(time_point<Clock, Duration2> const &timeStep):
  this constructor is defined as a member template, using the template header template <typename Duration2>. The type Duration2 is a std::chrono::duration (or comparable) type, using a possibly larger period for its unit than time_point’s Duration type. It initializes a time_point object to represent a point in time timeStep Duration2 units beyond the clock’s epoch.

The class std::chrono::time_point has these operators and members:

- std::chrono::time_point &operator+=(Duration const &duration):
  this operator is also available as binary arithmetic operator, expecting a std::chrono::time_point const & and a Duration const & argument (in any order). The amount of time represented by duration is added to the current time_point value. Example:

  ```cpp
  std::chrono::system_clock::now() + seconds(5);
  ```
• std::chrono::time_point &operator-=(Duration const &duration):
  this operator is also available as binary arithmetic operator, expecting a std::chrono::time_point const & and a Duration const & argument (in any order). The amount of time represented by duration is subtracted from the current time_point value. Example:
  auto point = std::chrono::system_clock::now();
  point -= seconds(5);
• constexpr Duration time_since_epoch() const:
  returns the object’s Duration since the epoch.
• static constexpr time_point min():
  a static member returning the value returned by the time_point’s duration::min value.
• static constexpr time_point max():
  a static member returning the value returned by the time_point’s duration::max value.

All predefined clocks use nanoseconds as their time unit. To obtain the time expressed in a larger time unit, divide the value returned by the time_point’s count value by larger time unit converted to nanoseconds. E.g., the number of hours passed since the beginning of the epoch is:

```cpp
using namespace std;
using namespace chrono; // for brevity

cout << system_clock::now().time_since_epoch().count() / nanoseconds(hours(1)).count() << " hours since the epoch\n";
```

### 20.1.5 Converting time to a NTBS

To convert time to a textual representation standard C functions can be used. These functions usually expect arguments in seconds, as returned by, e.g., the std::chrono::system_clock::to_time_t function. Standard C functions can be used to convert the returned time_t values to a textual representation. E.g.,

```cpp
using namespace std;
using namespace chrono; // for brevity

time_t tm = system_clock::to_time_t(system_clock::now() + hours(1));
cout << asctime(localtime(&tm));
```

Here are some additional examples showing how time_point objects can be used:

```cpp
#include <iostream>
#include <chrono>

using namespace std;
using namespace chrono;
```
```c++
int main()
{
    // the current time (or use 'auto')
    // 'now' is a time_point<system_clock>
    auto now(system_clock::now());
    // its value in seconds:
    cout << system_clock::to_time_t(now) << 'n';
    // now + two hours:
    cout << system_clock::to_time_t(now + hours(2)) << 'n';
    // define a time_point 1 hour after the epoch:
    time_point<system_clock> oneHrLater(hours(1));
    // show the epoch and the time in seconds of oneHrLater:
    cout << system_clock::to_time_t(time_point<system_clock>()) << ' ' <<
        system_clock::to_time_t(oneHrLater) << 'n';
}
```

### 20.2 Multi Threading

In C++ multi threading may be implemented at various levels of abstraction. In general the highest level of abstraction which is available to implement a mult-threaded problem should be used. Not so much because it’s often simpler than using lower levels of abstraction, but because higher levels of abstraction are usually semantically closer to the original problem description, resulting in code which is easier to understand and therefore easier to maintain. Also, high-abstraction classes also provide exception safety and prevent the occurrence of memory leaks.

C++‘s main tool for creating multi-threaded programs is the class `std::thread`, and some examples of its use have already been shown at the beginning of this chapter.

Characteristics of individual threads can be queried from the `std::this_thread` namespace. Also, `std::this_thread` offers some control over the behavior of an individual thread.

To synchronize access to shared data C++ offers mutexes (implemented by the class `std::mutex`) and condition variables (implemented by the class `std::condition_variable`).

Members of these classes may throw `system_error` objects (cf. section 10.9) when encountering a low-level error condition.

#### 20.2.1 The namespace `std::this_thread`

The namespace `std::this_thread` contains functions that are uniquely associated with the currently running thread.

Before using the namespace `this_thread` the `<thread>` header file must be included.

Inside the `std::this_thread` namespace several free functions are defined, providing information about the current thread or that can be used to control its behavior:
• thread::id this_thread::get_id() noexcept:
  returns an object of type thread::id that identifies the currently active thread of execution. For an active thread the returned id is unique in the sense that it maps 1:1 to the currently active thread, and is not returned by any other thread. If a thread is currently not running thread::id() is returned by the std::thread object's get_id member.

• void yield() noexcept:
  when a thread calls this_thread::yield() the current thread is briefly suspended, allowing other (waiting) threads to start.

• void sleep_for(chrono::duration<Rep, Period> const &relTime) noexcept:
  when a thread call this_thread::sleep_for(...) it is suspended for the amount of time that's specified in its argument. E.g.,
  ```cpp
  std::this_thread::sleep_for(std::chrono::seconds(5));
  ```

• void sleep_until(chrono::time_point<Clock, Duration> const &absTime) noexcept:
  when a thread calls this member it is suspended until the specified absTime is in the past. The next example has the same effect as the previous example:
  ```cpp
  // assume using namespace std
  this_thread::sleep_until(chrono::system_clock().now() + chrono::seconds(5));
  ```
  Conversely, the sleep_until call in the next example immediately returns:
  ```cpp
  this_thread::sleep_until(chrono::system_clock().now() - chrono::seconds(5));
  ```

### 20.2.2 The class std::thread

Multi threading in C++ starts off with objects of the class std::thread. Each object of this class handles a separate thread.

Before using Thread objects the <thread> header file must be included.

Thread objects can be constructed in various ways:

• thread() noexcept:
  The default constructor creates a thread object. As it receives no function to execute, it does not start a separate thread of execution. It is used, e.g., as a data member of a class, allowing class objects to start a separate thread at some later point in time;

• thread(thread &&tmp) noexcept:
  The move constructor takes ownership of the thread controlled by tmp, while tmp, if it runs a thread, loses control over its thread. Following this, tmp is in its default state, and the newly created thread is responsible for calling, e.g., join.

• explicit thread(Fun &&fun, Args &&...args):
  This member template (cf. section 22.1.3) expects a function (or functor) as its first argument. The function is immediately started as a separate thread. If the function (or functor) expects arguments, then these arguments can be passed to the thread’s
20.2. MULTI THREADING

constructor immediately following its first (function) argument. Additional arguments are passed with their proper types and values to fun. Following the thread object’s construction, a separately running thread of execution is started.

The notation Arg &&...args indicates that any additional arguments are passed as is to the function. The types of the arguments that are passed to the thread constructor and that are expected by the called function must match: values must be values, references must be reference, r-value references must be r-value references (or move construction must be supported). The following example illustrates this requirement:

```cpp
#include <iostream>
#include <thread>
using namespace std;

struct NoMove
{
  NoMove() = default;
  NoMove(NoMove &&tmp) = delete;
};

struct MoveOK
{
  int d_value = 10;
  MoveOK() = default;
  MoveOK(MoveOK const &) = default;
  MoveOK(MoveOK &&tmp)
  {
    d_value = 0;
    cout << "MoveOK move cons.\n";
  }
};

void valueArg(int value)
{
}

void refArg(int &ref)
{
}

void r_refArg(int &&tmp)
{
  tmp = 100;
}

void r_refNoMove(NoMove &&tmp)
{
}

void r_refMoveOK(MoveOK &&tmp)
{
}

int main()
{
  int value = 0;
  std::thread(valueArg, value).join();
  std::thread(refArg, ref(value)).join();
  std::thread(r_refArg, move(value)).join();
```

At lines 43 through 45 we see a value, reference, and an r-value reference being passed to a `std::thread`: with the functions running the threads expecting matching argument types.

- Line 47 fails to compile, as a value argument doesn’t match the reference expected by `refArg`. Note that this problem was solved in line 43 by using the `std::ref` function.

- On the other hand lines 49 and 58 compile OK, as `int` values and class-types supporting move operations can be passed as values to functions expecting r-value references. In this case notice that the functions expecting the r-value references do not access the provided arguments (except for the actions performed by their move constructors), but use move construction to create temporary values or objects on which the functions operate.

- Lines 52 and 55 won’t compile as the `NoMove` struct doesn’t offer a move constructor.

Be careful when passing local variables as arguments to thread objects: if the thread continues to run when the function whose local variables are used terminates, then the thread suddenly uses wild pointers or wild references, as the local variables no longer exist. To prevent this from happening (illustrated by the next example) do as follows:

- pass an anonymous copy of the local variable as argument to the thread constructor, or
- call `join` on the thread object to ensure that the thread has finished within the local variable’s lifetime.
15: std::thread safeLocal() {
16:     std::string text = "hello world";
17:     return std::thread(threadFun, std::string(text));
18: }
19:
20:
21: int main()
22: {
23:     std::thread local(safeLocal());
24:     std::cout << "safeLocal has ended\n";
25:     local.join();
26: }

In line 18 be sure not to call std::ref(text) instead of std::string(text).
If the thread cannot be created a std::system_error exception is thrown.
Since this constructor not only accepts functions but also function objects as its first argument, a local context may be passed to the function object’s constructor. Here is an example of a thread receiving a function object using a local context:

```cpp
#include <iostream>
#include <thread>
#include <array>

using namespace std;

class Functor {
    array<int, 30> &d_data;
    int d_value;

public:
    Functor(array<int, 30> &data, int value) :
        d_data(data),
        d_value(value)
    {}
    void operator()(ostream &out) {
        for (auto &value: d_data) {
            value = d_value++;
            out << value << ' ';
        }
        out << '\n';
    }
};

int main()
{
    array<int, 30> data;
    Functor functor(data, 5);
    thread funThread(functor, ref(cout));
    funThread.join();
};
```
The class `std::thread` does not provide a copy constructor.

The following members are available:

- **thread &operator=(thread &&tmp) noexcept**:

  If the operator's left-hand side operand (lhs) is a joinable thread, then `terminate` is called. Otherwise, `tmp` is assigned to the operator's lhs and `tmp`'s state is changed to the thread's default state (i.e., `thread()`).

- **void detach()**:

  Requires `joinable` (see below) to return true. The thread for which `detach` is called continues to run. The (e.g., parent) thread calling `detach` continues immediately beyond the `detach`-call. After calling `object.detach()`, 'object' no longer represents the (possibly still continuing but now detached) thread of execution. It is the detached thread's implementation's responsibility to release its resources when its execution ends.

  Since `detach` disconnects a thread from the running program, e.g., `main` no longer can wait for the thread's completion. As a program ends when `main` ends, its still running detached threads also stop, and a program may not properly finish all its threads, as demonstrated by the following example:

  ```cpp
  #include <thread>
  #include <iostream>
  #include <chrono>

  void fun(size_t count, char const *txt)
  {
      for (; count--; )
      {
          std::this_thread::sleep_for(std::chrono::milliseconds(100));
          std::cout << count << "": " << txt << std::endl;
      }
  }

  int main()
  {
      std::thread first(fun, 5, "hello world");
      first.detach();

      std::thread second(fun, 5, "a second thread");
      second.detach();

      std::this_thread::sleep_for(std::chrono::milliseconds(400));
      std::cout << "leaving" << std::endl;
  }
  ```

  A detached thread may very well continue to run after the function that launched it has finished. Here, too, you should be very careful not to pass local variables to the detached thread, as their references or pointers will be undefined once the function defining the local variables terminates:

  ```cpp
  #include <iostream>
  #include <thread>
  #include <chrono>
  ```
20.2. MULTI THREADING

```cpp
using namespace std;
using namespace chrono;

void add(int const &p1, int const &p2)
{
    this_thread::sleep_for(milliseconds(200));
    cerr << p1 << " + " << p2 << " = " << (p1 + p2) << \n';
}

void run()
{
    int v1 = 10;
    int v2 = 20;

    // thread(add, ref(v1), ref(v2)).detach(); // DON'T DO THIS
    thread(add, int(v1), int(v2)).detach(); // this is OK: own copies
}

void oops()
{
    int v1 = 0;
    int v2 = 0;
}

int main()
{
    run();
    oops();
    this_thread::sleep_for(seconds(1));
}
```

- `id get_id() const noexcept`:
  
  If the current object does not represent a running thread `thread::id()` is returned. Otherwise, the thread’s unique ID (also obtainable from with the thread from `this_thread::get_id()` is returned.

- `void join()`:
  
  Requires `joinable` to return `true`. Blocks the thread calling `join` until the thread for which `join` is called has completed. Following its completion the object whose `join` member was called no longer represents a running thread, and its `get_id` member will return `std::thread::id()`.

  This member was used in several examples shown so far. As noted: when `main` ends while a joinable thread is still running, `terminate` is called, aborting the program.

- `bool joinable() const noexcept`:
  
  Returns `object.get_id() != id()`, where `object` is the thread object for which `joinable` was called.

- `void swap(thread &other) noexcept`:
  
  The states of the thread object for which `swap` was called and `other` are swapped. Note that threads may always be swapped, even when their thread functions are currently being executed.
CHAPTER 20. MULTI THREADING

- unsigned thread::hardware_concurrency() noexcept:

  This static member returns the number of threads that can run at the same time on the current computer. On a stand-alone multi-core computer it (probably) returns the number of cores.

Things to note:

- When intending to define an anonymous thread it may appear not to start, unless you immediately also call join. E.g.,

```cpp
void doSomething();
int main()
{
    thread(doSomething);  // nothing happens??
    thread(doSomething).join()  // doSomething is executed??
}
```

This similar to the situation we encountered in section 7.5: the first statement doesn’t define an anonymous thread object at all. It simply defines the thread object doSomething. Consequently, compilation of the second statement fails, as there is no thread(thread &) constructor. When the first statement is omitted, the doSomething function is executed by the second statement. If the second statement is omitted, a default constructed thread object by the name of doSomething is defined.

- A thread only starts after its construction has completed. This includes move constructions or move assignments. E.g., in a statement like

```cpp
    thread object(thread(doSomething));
```

the move constructor is used to transfer control from an anonymous thread executing doSomething to the thread object. Only after object’s construction has completed doSomething is started in the separate thread.

- Exceptions thrown from the thread (e.g., by the function defining the thread’s actions) are local to the executed thread. Either they must be caught by the executing thread (as each running thread has its own execution stack), or they can be passed to the starting thread using a packaged_task and a future (cf., respectively, sections 20.12 and 20.9).

A thread ends when the function executing a thread finishes. When a thread object is destroyed while its thread function is still running, terminate is called, aborting the program’s end. Bad news: the destructors of existing objects aren’t called and exceptions that are thrown are left uncaught. This happens in the following program as the thread is still active when main ends:

```cpp
#include <iostream>
#include <thread>

void hello()
{
    while (true)
    {
        std::cout << "hello world!\n";
    }
}

int main()
{
    std::thread hi(hello);
}
```
There are several ways to solve this problem. One of them is discussed in the next section.

### 20.2.2.1 Static data and threads: std::thread_local

With multi-threaded programs the well-known distinction between global and local data is somewhat too coarse. For single- and multi-threaded programs alike, global data are available to all of the program's code, and local data are available to the function (or compound statement) in which the local data are defined. But multi-threaded programs may feel the need for an intermediate type of data, uniquely available to the different threads.

The `thread_local` keyword provides this intermediate data level. Global variables declared as `thread_local` are global within each individual thread. Each thread owns a copy of the `thread_local` variables, and may modify them at will. A `thread_local` variable in one thread is completely separated from that variable in another thread. Here is an example:

```cpp
#include <iostream>
#include <thread>

using namespace std;

thread_local int t_value = 100;

void modify(char const *label, int newValue)
{
  cout << label << " before: " << t_value << ". Address: " <<
       &t_value << endl;
  t_value = newValue;
  cout << label << " after: " << t_value << endl;
}

int main()
{
  thread(modify, "first", 50).join();
  thread(modify, "second", 20).join();
  modify("main", 0);
}
```

- At line 6 the `thread_local` variable `t_value` is defined. It is initialized to 100, and that becomes the initial value for each separately running thread;
- In lines 8 through 14 the function `modify` is defined. It assigns a new value to `t_value`;
- At lines 18 and 19 two threads are started, which are immediately joining the main thread again.
- The main thread itself is also a thread, and it directly calls `modify`.

Running this program shows that each separate thread starts with `t_value` being 100, and then modifies it without affecting the values of `t_value` used by other threads.

Note that, although the `t_value` variables are unique to each thread, identical addresses may be shown for them. Since each thread uses its own stack, these variables may occupy the same relative locations within their respective stacks, giving the illusion that their physical addresses are identical.
20.2.2.2 Exceptions and join()

Once a thread starts and it isn’t detached it must eventually join its starting (parent) thread, or the program aborts. Usually, once a thread has started the parent thread continues to do some work by itself:

```cpp
void childActions();
void doSomeWork();

void parent()
{
    thread child(childActions);
    doSomeWork();
    child.join();
}
```

However, maybe `doSomeWork` can’t complete its work, and throws an exception, to be caught outside of `parent`. This, unfortunately, ends `parent`, and `child.join()` is missed. Consequently, the program aborts because of a thread that hasn’t been joined.

Clearly, all exceptions must be caught, `join` must be called, and the exception must be rethrown. But `parent` cannot use a function try-block, as the thread object is already out of scope once execution reaches the matching catch-clause. So we get:

```cpp
void childActions();
void doSomeWork();

void parent()
{
    thread child(childActions);
    try
    {
        doSomeWork();
        child.join();
    }
    catch (...)
    {
        child.join();
        throw;
    }
}
```

This is ugly: suddenly the function’s code is clobbered with a `try-catch` clause, as well as some unwelcome code-duplication.

This situation can be avoided using object based programming. Like, e.g., unique pointers, which use their destructors to encapsulate the destruction of dynamically allocated memory, we can use a comparable technique to encapsulate thread joining in an object’s destructor.

By defining the `thread` object inside a class we’re sure that by the time the our object goes out of scope, even if the `childActions` function throws an exception, the thread’s `join` member is called. Here are the bare essentials of our `JoinGuard` class, providing the join-guarantee (using in-line member implementations for brevity):

```cpp
1: #include <thread>
```
2. class JoinGuard
3: {
4: std::thread d_thread;
5: public:
6: JoinGuard(std::thread &&threadObj)
7: d_thread(std::move(threadObj))
8: {}
9: ~JoinGuard()
10: {
11: if (d_thread.joinable())
12: d_thread.join();
13: }
14: }
15: }
16: }
17:);

- At line 3 \texttt{childActions} is declared. Its implementation defines the child thread's actions.
- At line 8 its only constructor starts: it receives a temporary \texttt{thread} object, which is moved, in line 10, to \texttt{JoinGuard}'s \texttt{d\_thread} data member.
- When the \texttt{JoinGuard} object ceases to exist, its destructor (line 12) makes sure the thread is joined if it's still joinable (lines 14 and 15).

Here is an example how \texttt{JoinGuard} could be used:

1: #include <iostream>
2: #include "joinguard.h"
3: void childActions();
4: void doSomeWork();
5: void parent()
6: {
7: JoinGuard(std::thread(childActions));
8: doSomeWork();
9: }
10: void main()
11: try
12: {
13: parent();
14: }
15: catch (std::exception const &exc)
16: {
17: std::cout << exc.what() << '\n';
18: }

- The \texttt{main} function (lines 17 through 25) provides the function try-block to catch the exception thrown by \texttt{parent};
The parent function defines (line 13) an anonymous JoinGuard, receiving an anonymous thread object. Anonymous objects are used, as the parent function doesn’t need to access them anymore.

In line 14 doSomeWork is called, which throws an exception. This ends parent, but just before that JoinGuard’s destructor makes sure that the child-thread has been joined.

### 20.3 Synchronization (mutexes)

The C++11 standard offers mutex classes to protect shared data.

Before using mutexes the `<mutex>` header file must be included.

One of the key characteristics of multi-threaded programs is that threads may share data. Functions running as separate threads have access to all global data, and may also share the local data of their parent threads. However, unless proper measures are taken, this may easily result in data corruption, as illustrated by the following simulation of some steps that could be encountered in a multi-threaded program:

<table>
<thead>
<tr>
<th>Time step</th>
<th>Thread 1:</th>
<th>Thread 2:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>starts</td>
<td></td>
<td>T1 active</td>
</tr>
<tr>
<td>2</td>
<td>writes var</td>
<td></td>
<td>T1 commences writing</td>
</tr>
<tr>
<td>3</td>
<td>stopped</td>
<td></td>
<td>Context switch</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>starts</td>
<td>T2 active</td>
</tr>
<tr>
<td>5</td>
<td>writes var</td>
<td></td>
<td>T2 commences writing</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>assigns 10</td>
<td>T2 writes 10</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>stopped</td>
<td>Context switch</td>
</tr>
<tr>
<td>8</td>
<td>assigns 12</td>
<td></td>
<td>T1 writes 12</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

In this example, threads 1 and 2 share variable `var`, initially having the value 5. At step 1 thread 1 starts, and starts to write a value into `var`. However, it is interrupted by a context switch, and thread 2 is started (step 4). Thread 2 also wants to write a value into `var`, and succeeds until time step 7, when another context switch takes place. By now `var` is 10. However, thread 1 was also in the process of writing a value into `var`, and it is given a chance to complete its work: it assigns 12 to `var` in time step 8. Once time step 9 is reached, thread 2 proceeds on the (erroneous) assumption that `var` must be equal to 10. Clearly, from the point of view of thread 2 its data have been corrupted.

In this case data corruption was caused by multiple threads accessing the same data in an uncontrolled way. To prevent this from happening, access to shared data should be protected in such a way that only one thread at a time may access the shared data.

**Mutexes** are used to prevent the abovementioned kinds of problems by offering a guarantee that data are only accessed by the thread that could lock the mutex that is used to synchronize access to those data.

Exclusive data access completely depends on cooperation between the threads. If thread 1 uses mutexes, but thread 2 doesn’t, then thread 2 may freely access the common data. Of course that’s bad practice, which should be avoided.
It is stressed that although using mutexes is the programmer’s responsibility, their implementation isn’t: mutexes offer the necessary atomic calls. When requesting a mutex-lock the thread is blocked (i.e., the mutex statement does not return) until the lock has been obtained by the requesting thread.

Apart from the class `std::mutex` the class `std::recursive_mutex` is available. When a `recursive_mutex` is called multiple times by the same thread it increases its lock-count. Before other threads may access the protected data the recursive mutex must be unlocked again that number of times. Moreover, the classes `std::timed_mutex` and `std::recursive_timed_mutex` are available. Their locks expire when released, but also after a certain amount of time.

The members of the mutex classes perform atomic actions: no context switch occurs while they are active. So when two threads are trying to lock a mutex only one can succeed. In the above example: if both threads would use a mutex to control access to `var` thread 2 would not have been able to assign 12 to `var`, with thread 1 assuming that its value was 10. We could even have two threads running purely parallel (e.g., on two separate cores). E.g.:

<table>
<thead>
<tr>
<th>Time step:</th>
<th>Thread 1:</th>
<th>Thread 2:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>starts</td>
<td>starts</td>
<td>T1 and T2 active</td>
</tr>
<tr>
<td>2</td>
<td>locks</td>
<td>locks</td>
<td>Both threads try to lock the mutex</td>
</tr>
<tr>
<td>3</td>
<td>blocks...</td>
<td>obtains lock</td>
<td>T2 obtains the lock, and T1 must wait</td>
</tr>
<tr>
<td>4</td>
<td>(blocked)</td>
<td>processes var</td>
<td>T2 processes var, T1 still blocked</td>
</tr>
<tr>
<td>5</td>
<td>obtains lock</td>
<td>releases lock</td>
<td>T2 releases the lock, and T1 immediately obtains the lock</td>
</tr>
<tr>
<td>6</td>
<td>processes var</td>
<td>now T1 processes var</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>releases lock</td>
<td>T1 also releases the lock</td>
<td></td>
</tr>
</tbody>
</table>

Although mutexes can directly be used in programs, this rarely happens. It is more common to embed mutex handling in locking classes that make sure that the mutex is automatically unlocked again when the mutex lock is no longer needed. Therefore, this section merely offers an overview of the interfaces of the mutex classes. Examples of their use will be given in the upcoming sections (e.g., section 20.4).

All mutex classes offer the following constructors and members:

- **mutex() constexpr**:
  
The default constexpr constructor is the only available constructor;

- **~mutex()**:
  
The destructor does not unlock a locked mutex. If locked it must explicitly be unlocked using the mutex’s `unlock` member;

- **void lock()**:
  
  the calling thread blocks until it owns the mutex. Unless `lock` is called for a recursive mutex a `system_error` is thrown if the thread already owns the lock. Recursive mutexes increment their internal `lock count`;
• **bool try_lock() noexcept:**
  the calling thread tries to obtain ownership of the mutex. If ownership is obtained, `true` is returned, otherwise `false`. If the calling thread already owns the lock `true` is also returned, and in this case a recursive mutex also increments its internal `lock count`;

• **void unlock() noexcept:**
  the calling thread releases ownership of the mutex. A `system_error` is thrown if the thread does not own the lock. A recursive mutex decrements its internal lock count, releasing ownership of the mutex once the lock count has decayed to zero;

The **timed-mutex classes** (`timed_mutex`, `recursive_timed_mutex`) also offer these members:

• **bool try_lock_for(chrono::duration<Rep, Period> const &relTime) noexcept:**
  the calling thread tries to obtain ownership of the mutex within the specified time interval. If ownership is obtained, `true` is returned, otherwise `false`. If the calling thread already owns the lock `true` is also returned, and in this case a recursive timed mutex also increments its internal `lock count`. The `Rep` and `Duration` types are inferred from the actual `relTime` argument. E.g.,
  ```cpp
  std::timed_mutex timedMutex;
  timedMutex.try_lock_for(chrono::seconds(5));
  ```

• **bool try_lock_until(chrono::time_point<Clock, Duration> const &absTime) noexcept:**
  the calling thread tries to obtain ownership of the mutex until `absTime` has passed. If ownership is obtained, `true` is returned, otherwise `false`. If the calling thread already owns the lock `true` is also returned, and in this case a recursive timed mutex also increments its internal `lock count`. The `Clock` and `Duration` types are inferred from the actual `absTime` argument. E.g.,
  ```cpp
  std::timed_mutex timedMutex;
  timedMutex.try_lock_until(chrono::system_clock::now() + chrono::seconds(5));
  ```

### 20.3.1 Initialization in multi-threaded programs

Before using the `std::once_flag` and the `std::call_once` function, introduced in this section, the `<mutex>` header file must be included.

In single threaded programs the initialization of global data not necessarily happens at the same point in code. An example is the initialization of the object of a singleton class (cf. *Gamma et al.* (1995), Design Patterns, Addison-Wesley). Singleton classes may define a single static pointer data member `Singleton *s_object`, pointing to the singleton’s object, and may offer a static member `instance`, implemented something like this:

```cpp
Singleton &Singleton::instance()
{
    return s_object ?
        s_object :
        (s_object = new Singleton);
}
```
With multi-threaded programs this approach immediately gets complex. For example, if two threads call `instance` at the same time, while `s_object` still equals 0, then both may call `new Singleton`, resulting in one dynamically allocated `Singleton` object becoming unreachable. Other threads, called after `s_object` was initialized for the first time, may either return a reference to that object, or may return a reference to the object initialized by the second thread. Not exactly the expected behavior of a singleton.

Mutexes (cf. section 20.3) can be used to solve these kinds of problems, but they result in some overhead and inefficiency, as the mutex must be inspected at each call of `Singleton::instance`.

When variables must dynamically be initialized, and the initialization should take place only once the C++11 standards offers a better alternative in the form of the `std::once_flag` type and the `std::call_once` function.

The `call_once` function expects two or three arguments:

- The first argument is a `once_flag` variable, keeping track of the actual initialization status. The `call_once` function simply returns if the `once_flag` indicates that initialization already took place;
- The second argument is the address of a function which must be called only once. This function may be a free function or it may be the address of a class member function;
- If the second argument is the address of a class member function, then the object for which the member function should be called must be provided as `call_once`'s third argument.

A thread-safe implementation of the singleton's `instance` function can now easily be designed (using in-class implementations for brevity):

```cpp
class Singleton
{
    static std::once_flag s_once;
    static Singleton *s_singleton;
    ...
    public:
    static Singleton *instance()
    {
        std::call_once(s_once, []{s_singleton = new Singleton;});
        return s_singleton;
    }
    ...
};
```

However, there are additional ways to initialize data, even for multi-threaded programs:

- First, suppose a constructor is declared with the `constexpr` keyword (cf. section 8.1.4.1), satisfying the requirements for constant initialization. In this case, a static object, initialized using that constructor, is guaranteed to be initialized before any code is run as part of the static initialization phase. This used by `std::mutex`, as it eliminates the possibility of race conditions when global mutexes are initialized.
- Second, a static variable defined within a compound statement may be used (e.g., a static local variable within a function body). Static variables defined within a compound statement are initialized the first time the function is called at the point in the code where the static variable is defined. Here is an example:
#include <iostream>

struct Cons
{
    Cons()
    {
        std::cout << "Cons called\n";
    }
}

void called(char const*time)
{
    std::cout << time << "time called() activated\n";
    static Cons cons;
}

int main()
{
    std::cout << "Pre-1\n";
    called("first");
    called("second");
    std::cout << "Pre-2\n";
    Cons cons;
}

/*
 * Displays:
 * Pre-1
 * firsttime called() activated
 * Cons called
 * seconddtime called() activated
 * Pre-2
 * Cons called
 */

This feature causes a thread to wait automatically if another thread is still initializing the static data (note that non-static data never cause problems, as non-static local variables only exist within their own thread of execution).

## 20.4 Locks and lock handling

Locks are used to simplify the use of mutexes. Before locks can be used the `<mutex>` header file must be included.

Whenever threads share data, and at least one of the threads may change common data, mutexes should be used to prevent threads from using the same data synchronously.

Usually locks are released at the end of action blocks. This requires explicit calls to the mutexes’ `unlock` function, with introduces comparable problems as we’ve seen with the thread’s `join` member.

To simplify locking and unlocking the C++11 standard provides two mutex wrapper classes:

- `std::lock_guard`:
  
  objects of this class offer the basic unlock-guarantee: their destructors call the member `unlock` of the mutexes they control;
20.4. LOCKS AND LOCK HANDLING

- std::unique_lock:
  objects of this class offer a more extensive interface, allowing explicit unlocking and locking of the mutexes they control, while their destructors preserve the unlock-guarantee also offered by lock_guard;

The class lock_guard offers a limited, but useful interface:

- lock_guard<Mutex>(Mutex &mutex):
  when defining a lock_guard object the mutex type (e.g., std::mutex, std::timed_mutex) is specified, and a mutex of the indicated type is provided as its argument. The construction blocks until the lock_guard object owns the lock. The lock_guard’s destructor automatically releases the mutex lock.

- lock_guard<Mutex>(Mutex &mutex, std::adopt_lock_t):
  this constructor is used to transfer control over the mutex from the calling thread to the lock_guard. The mutex lock is released again by the lock_guard’s destructor. At construction time the mutex must already be owned by the calling thread. Here is an illustration of how it can be used:

```cpp
1: void threadAction(std::mutex &mut, int &sharedInt)  
2: {  
3:   std::lock_guard<std::mutex>{mut, std::adopt_lock_t()};  
4:   // do something with sharedInt  
5: }  
```
- At line 1 threadAction receives a reference to a mutex. Assume the mutex owns the lock;
- At line 3 control is transferred to the lock_guard. Since we don’t explicitly use the lock_guard object, an anonymous object can be defined;
- When the function ends, at line 5, the mutex’s lock is released.

- mutex_type:
  in addition to the constructors and destructor, lock_guard<Mutex> types also define the type mutex_type: it is a synonym of the Mutex type that is passed to the lock_guard’s constructor.

Here is a simple example of a multi-threaded program using lock_guards to prevent information inserted into cout from getting mixed.

```cpp
bool oneLine(istream &in, mutex &mut, int nr)  
{  
  lock_guard<mutex> lg(mut);  
  string line;  
  if (not getline(in, line))  
    return false;  
  cout << nr << ": " << line << endl;  
  return true;  
}
```
void io(istream &in, mutex &mut, int nr)
{
    while (oneLine(in, mut, nr))
        this_thread::yield();
}

int main(int argc, char **argv)
{
    ifstream in(argv[1]);
    mutex ioMutex;

    thread t1(io, ref(in), ref(ioMutex), 1);
    thread t2(io, ref(in), ref(ioMutex), 2);
    thread t3(io, ref(in), ref(ioMutex), 3);

    t1.join();
    t2.join();
    t3.join();
}

As with lock_guard, a mutex-type must be specified when defining objects of the class std::unique_lock. The class unique_lock is much more elaborate than the basic lock_guard class template. Its interface does not define a copy constructor or overloaded assignment operator, but it does define a move constructor and a move assignment operator. In the following overview of unique_lock's interface Mutex refers to the mutex-type that is specified when defining a unique_lock:

- unique_lock() noexcept:
  
  the default constructor is not yet associated with a mutex object. It must be assigned a mutex (e.g., using move-assignment) before it can do anything useful;

- explicit unique_lock(Mutex &mutex):

  initializes a unique_lock with an existing Mutex object, and calls mutex.lock();

- unique_lock(Mutex &mutex, defer_lock_t) noexcept:

  initializes a unique_lock with an existing Mutex object, but does not call mutex.lock(). Call it by passing a defer_lock_t object as the constructor's second argument, e.g.,

  unique_lock<mutex> ul(mutexObj, defer_lock_t())  // or, anonymous:
  unique_lock<mutex>{mutexObj, defer_lock_t()}

- unique_lock(Mutex &mutex, try_to_lock_t) noexcept:

  initializes a unique_lock with an existing Mutex object, and calls mutex.try_lock(): the constructor won't block if the mutex cannot be locked;

- unique_lock(Mutex &mutex, adopt_lock_t) noexcept:

  initializes a unique_lock with an existing Mutex object, and assumes that the current thread has already locked the mutex;
• `unique_lock(Mutex &mutex, chrono::duration<Rep, Period> const &relTime)` 
  noexcept:
  
  this constructor tries to obtain ownership of the `Mutex` object by calling 
  `mutex.try_lock_for(relTime)`. The specified mutex type must therefore sup-
  port this member (e.g., it is a `std::timed_mutex`). It could be called like this:
  
  ```cpp
  std::unique_lock<std::timed_mutex> ulock(timedMutex,
                                          std::chrono::seconds(5));
  ```

• `unique_lock(Mutex &mutex, chrono::time_point<Clock, Duration> const &absTime)` 
  noexcept:
  
  this constructor tries to obtain ownership of the `Mutex` object by calling 
  `mutex.try_lock_until(absTime)`. The specified mutex type must therefore sup-
  port this member (e.g., it is a `std::timed_mutex`). This constructor could be called 
  like this:
  
  ```cpp
  std::unique_lock<std::timed_mutex> ulock(
      timedMutex,
      std::chrono::system_clock::now() + std::chrono::seconds(5)
  );
  ```

• `void lock()`:
  
  blocks the current thread until ownership of the mutex that is managed by the 
  `unique_lock` is obtained. If no mutex is currently managed, then a `system_error` 
  exception is thrown.

• `Mutex *mutex() const noexcept`:
  
  returns a pointer to the mutex object stored inside the `unique_lock` (a `nullptr` is 
  returned if no mutex object is currently associated with the `unique_lock` object.)

• `explicit operator bool() const noexcept`:
  
  returns `true` if the `unique_lock` owns a locked mutex, otherwise `false` is returned;

• `unique_lock& operator=(unique_lock &&tmp)` noexcept:
  
  if the left-hand operand owns a lock, it will call its mutex’s `unlock` member, where-
  after `tmp`’s state is transferred to the left-hand operand;

• `bool owns_lock()` const noexcept:
  
  returns `true` if the `unique_lock` owns the mutex, otherwise `false` is returned;

• `Mutex *release()` noexcept:
  
  returns a pointer to the mutex object that is associated with the `unique_lock` object, 
  discarding that association;

• `void swap(unique_lock& other)` noexcept:
  
  swaps the states of the current `unique_lock` and `other`;

• `bool try_lock()`:
  
  tries to obtain ownership of the mutex that is associated with the `unique_lock`, 
  returning `true` if this succeeds, and `false` otherwise. If no mutex is currently asso-
  ciated with the `unique_lock` object, then a `system_error` exception is thrown;
• `bool try_lock_for(chrono::duration<Rep, Period> const &relTime)`:
  this member function tries to obtain ownership of the Mutex object managed by
  the unique_lock object by calling the mutex's try_lock_for(relTime) member. The specified mutex type must therefore support this member (e.g., it is a
  `std::timed_mutex`);

• `bool try_lock_until(chrono::time_point<Clock, Duration> const &absTime)`:
  this member function tries to obtain ownership of the Mutex object managed by
  the unique_lock object by calling the mutex's `mutex.try_lock_until(absTime)`
  member. The specified mutex type must therefore support this member (e.g., it is a
  `std::timed_mutex`);

• `void unlock()`:
  releases ownership of the mutex (or reduces the mutex's lock count). A
  `system_error` exception is thrown if the unique_lock object does not own the mu-
  tex.

In addition to the members of the classes `std::lock_guard` and `std::unique_lock` the functions
`std::lock` and `std::try_lock` are available. These functions can be used to prevent deadlocks,
the topic of the next section.

### 20.4.1 Deadlocks

A deadlock occurs when two locks are required to process data, but one thread obtains the first lock
and another thread obtains the second lock. The C++11 standard defines the generic `std::lock` and
`std::try_lock` functions that can be used to help preventing such situations.

Before these functions can be used the `<mutex>` header file must be included

In the following overview `L1 &l1, ...` represents one or more references to objects of lockable
types:

• `void std::lock(L1 &l1, ...)`:  
  When the function returns locks were obtained on all `l1` objects. If a lock could not
  be obtained for at least one of the objects, then all locks obtained so far are released,
  even if the object for which no lock could be obtained threw an exception;

• `int std::try_lock(L1 &l1, ...)`:  
  This function calls the lockable objects' `try_lock` members. If all locks could be
  obtained, then -1 is returned. Otherwise the (0-based) index of the first argument
  which could not be locked is returned, releasing all previously obtained locks.

As an example consider the following little multi-threaded program: The threads use mutexes to
obtain unique access to `cout` and to an `int` `value`. However, `fun1` first locks `cout` (line 7), and
then `value` (line 10); `fun2` first locks `value` (line 16) and then `cout` (line 19). Clearly, if `fun1`
has locked `cout` `fun2` can’t obtain the lock until `fun1` has released it. Unfortunately, `fun2` has
locked `value`, and the functions only release their locks when returning. But in order to access
the information in `value` `fun1` it must have obtained a lock on `value`, which it can’t, as `fun2` has
already locked `value`: the threads are waiting for each other, and neither thread gives in.

```java
1: int value;
```
A good recipe for avoiding deadlocks is to prevent nested (or multiple) mutex lock calls. But if multiple mutexes must be used, always obtain the locks in the same order. Rather than doing this yourself, `std::lock` and `std::try_lock` should be used whenever possible to obtain multiple mutex locks. These functions accept multiple arguments, which must be lockable types like `lock_guard`, `unique_lock`, or even a plain `mutex`. The previous deadlocking program, can be modified to call `std::lock` to lock both mutexes. In this example using one single mutex would also work, but the modified program now looks as similar as possible to the previous program. Note how in lines 10 and 21 a different ordering of the `unique_locks` arguments was used: it is not necessary to use an identical argument order when calling `std::lock` or `std::try_lock`.

```
int value;
mutex valueMutex;
mutex coutMutex;

void fun1()
{
   unique_lock<mutex> lg1(coutMutex, defer_lock);
   unique_lock<mutex> lg2(valueMutex, defer_lock);
   lock(lg1, lg2);
   cout << "fun 1 locks cout\n";
   cout << "fun 1 locks value\n";
}

void fun2()
```
20.5 Event handling (condition variables)

This section introduces condition variables. Condition variables allow programs to synchronize threads using the states of data, rather than simply locking the access to data (which is realized using mutexes).

Before condition variables can be used the <condition_variable> header file must be included.

To start our discussion, consider a classic producer-consumer scenario: the producer generates items which are consumed by a consumer. The producer can only produce a certain number of items before its storage capacity has filled up and the client cannot consume more items than the producer has produced.

At some point the producer's storage capacity has filled to the brim, and the producer has to wait until the client has at least consumed some items, thereby creating space in the producer's storage. Similarly, the consumer cannot start consuming until the producer has at least produced some items.

Implementing this scenario only using mutexes (data locking) is not an attractive option, as merely using mutexes forces a program to implement the scenario using polling: processes must continuously (re)acquire the mutex's lock, determine whether they can perform some action, followed by the release of the lock. Often there's no action to perform, and the process is busy acquiring and releasing the mutex's lock. Polling forces threads to wait until they can lock the mutex, even though continuation might already be possible. The polling interval could be reduced, but that too isn't an attractive option, as that increases the overhead associated with handling the mutexes (also called 'busy waiting').

Condition variables can be used to prevent polling. Threads can use condition variables to notify waiting threads that there is something for them to do. This way threads can synchronize on data values (states).

As data values may be modified by multiple threads, threads still need to use mutexes, but only for controlling access to the data. In addition, condition variables allow threads to release ownership of mutexes until a certain value has been obtained, until a preset amount of time has been passed, or until a preset point in time has been reached.

The prototypical setup of threads using condition variables looks like this:

```cpp
unique_lock<mutex> lg1(coutMutex, defer_lock);
unique_lock<mutex> lg2(valueMutex, defer_lock);
lock(lg2, lg1);
cout << "fun 2 locks cout\n";
cout << "fun 2 locks value\n";
}

int main()
{
    thread t1(fun1);
    thread t2(fun2);
t1.join();
t2.join();
}
```
• consumer thread(s) act like this:

    lock the mutex
    while the required condition has not yet been attained (i.e., is false):
        wait until being notified
            (this automatically releasing the mutex’s lock).
    once the mutex’s lock has been reacquired, and the required condition
    has been attained:
        process the data
    release the mutex’s lock.

• producer thread(s) act similarly:

    lock the mutex
    while the required condition has not yet been attained:
        do something to attain the required condition
        notify waiting threads (that the required condition has been attained)
    release the mutex’s lock.

No matter which thread starts, the thread holding the mutex’s lock will at some point release the
lock, allowing the other process to (re)acquire it. If the consumer starts it immediately releases the
lock once it enters its waiting state; if the producer starts it releases the lock once the condition is
true.

This protocol hides a subtle initial synchronization requirement. The consumer will miss the pro-
ducer’s notification if it (i.e., the consumer) hasn’t yet entered its waiting state. So waiting (con-
sumer) threads should start before notifying (producer) threads. Once threads have started, no as-
sumptions can be made anymore about the order in which any of the condition variable’s members
(notify_one, notify_all, wait, wait_for, and wait_until) are called.

Condition variables come in two flavors: objects of the class `std::condition_variable` are used
in combination with objects of type `unique_lock<mutex>`. Because of optimizations which are
available for this specific combination using `condition_variable` is somewhat more efficient
than using the more generally applicable class `std::condition_variable_any`, which may be
used with any (e.g., user supplied) lock type.

Condition variable classes (covered in detail in the next two sections) offer members like `wait`,
`wait_for`, `wait_until`, `notify_one` and `notify_all` that may concurrently be called. The
notifying members are always atomically executed. Execution of the `wait` members consists of
three atomic parts:

• the mutex is released, and the thread is suspended until its notification;
• Once the notification has been received, the lock is reacquired
• The wait state ends (and processing continues beyond the `wait` call).

So, returning from `wait`-members the previously waiting thread has reacquired the mutex’s lock.

In addition to the condition variable classes the following free function and `enum` type is provided:

• `void std::notify_all_at_thread_exit(condition_variable &cond,
    unique_lock<mutex> lockObject);

    once the current thread has ended, all other threads waiting on `cond` are no-
    tified. It is good practice to exit the thread as soon as possible after calling
    `notify_all_at_thread_exit`. 
Waiting threads must verify that the thread they were waiting for has indeed ended. This is usually realized by first obtaining the lock on lockObject, followed by verifying that the condition they were waiting for is true and that the lock was not reacquired before notify_all_at_thread_exit was called.

- **std::cv_status:**
  
  the cv_status enum is used by several member functions of the condition variable classes (cf. sections 20.5.1 and 20.5.2):

  namespace std
  {
      enum class cv_status
      {
          no_timeout,
          timeout
      };
  }

20.5.1 The class std::condition_variable

The class std::condition_variable merely offers a default constructor. No copy constructor or overloaded assignment operator is provided.

Before using the class condition_variable the <condition_variable> header file must be included.

The class's destructor requires that no thread is blocked by the thread destroying the condition_variable. So all threads waiting on a condition_variable must be notified before a condition_variable object's lifetime ends. Calling notify_all (see below) before a condition_variable’s lifetime ends ensures care of that, as the condition_variable’s thread releases its lock of the mutex variable, allowing one of the notified threads to lock the mutex.

In the following member-descriptions a type Predicate indicates that a provided Predicate argument can be called as a function without arguments, returning a bool. Also, other member functions are frequently referred to. It is tacitly assumed that all member referred to below were called using the same condition variable object.

The class condition_variable supports several wait members, which block the thread until notified by another thread (or after a configurable waiting time). However, wait members may also spuriously unblock, without having reacquired the lock. Therefore, returning from wait members threads should verify that the required condition is actually true. If not, again calling wait may be appropriate. The next piece of pseudo code illustrates this scheme:

```cpp
while (conditionNotTrue())
    condVariable.wait(&uniqueLock);
```

The class condition_variable's members are:

- **void notify_one() noexcept:**

  one wait member called by other threads returns. Which one actually returns cannot be predicted.
• void notify_all() noexcept:
  all wait members called by other threads unblock their wait states. Of course, only one of them will subsequently succeed in reacquiring the condition variable’s lock object.

• void wait(unique_lock<mutex>& uniqueLock):
  before calling wait the current thread must have acquired the lock of uniqueLock. Calling wait releases the lock, and the current thread is blocked until it has received a notification from another thread, and has reacquired the lock.

• void wait(unique_lock<mutex>& uniqueLock, Predicate pred):
  this is a member template, using the template header template <typename Predicate>. The template’s type is automatically derived from the function’s argument type and does not have to be specified explicitly.
  Before calling wait the current thread must have acquired the lock of uniqueLock. As long as ‘pred’ returns false wait(lock) is called.

• cv_status wait_for(unique_lock<mutex> &uniqueLock, std::chrono::duration<Rep, Period> const &relTime):
  this member is defined as a member template, using the template header template <typename Rep, typename Period>. The template’s types are automatically derived from the types of the function’s arguments and do not have to be specified explicitly. E.g., to wait for at most 5 seconds wait_for can be called like this:
  cond.wait_for(&unique_lock, std::chrono::seconds(5));
  This member returns when being notified or when the time interval specified by relTime has passed.
  When returning due to a timeout, std::cv_status::timeout is returned, otherwise std::cv_status::no_timeout is returned.
  Threads should verify that the required data condition has been met after wait_for has returned.

• bool wait_for(unique_lock<mutex> &uniqueLock, std::chrono::duration<Rep, Period> const &relTime, Predicate pred):
  this member is defined as a member template, using the template header template <typename Rep, typename Period, typename Predicate>. The template’s types are automatically derived from the types of the function’s arguments and do not have to be specified explicitly.
  As long as pred returns false, the previous wait_for member is called. If the previous member returns cv_status::timeout, then pred is returned, otherwise true.

• cv_status wait_until(unique_lock<mutex>& uniqueLock, std::chrono::time_point<Clock, Duration> const &absTime):
  this member is defined as a member template, using the template header template <typename Clock, typename Duration>. The template’s types are automatically derived from the types of the function’s arguments and do not have to be specified explicitly. E.g., to wait until 5 minutes after the current time wait_until can be called like this:
  cond.wait_until(&unique_lock, std::chrono::system_clock::now() +
  std::chrono::minutes(5));
CHAPTER 20. MULTI THREADING

This function acts identically to the `wait_for(unique_lock<mutex> &uniqueLock, chrono::duration<Rep, Period> const &relTime)` member described earlier, but uses an absolute point in time, rather than a relative time specification.

This member returns when being notified or when the time interval specified by `relTime` has passed. When returning due to a timeout, `std::cv_status::timeout` is returned, otherwise `std::cv_status::no_timeout` is returned.

- `bool wait_until(unique_lock<mutex> &lock, chrono::time_point<Clock, Duration> const &absTime, Predicate pred)`:

  this member is defined as a member template, using the template header `<template <typename Clock, typename Duration, typename Predicate>`. The template's types are automatically derived from the types of the function's arguments and do not have to be specified explicitly. As long as `pred` returns false, the previous `wait_until` member is called. If the previous member returns `cv_status::timeout`, then `pred` is returned, otherwise `true`.

Threads should verify that the required condition is `true` when `wait`-members of condition variables return.

### 20.5.2 The class std::condition_variable_any

Different from the class `condition_variable` the class `std::condition_variable_any` can be used with any (e.g., user supplied) lock type, and not just with the stl-provided `unique_lock<mutex>`.

Before using the class `condition_variable_any` the `<condition_variable>` header file must be included.

The functionality that is offered by `condition_variable_any` is identical to the functionality offered by the class `condition_variable`, albeit that the lock-type that is used by `condition_variable_any` is not predefined. The class `condition_variable_any` therefore requires the specification of the lock-type that must be used by its objects.

In the interface shown below this lock-type is referred to as `Lock`. Most of `condition_variable_any`'s members are defined as member templates, defining a `Lock` type as one of its parameters. The requirements of these lock-types are identical to those of the stl-provided `unique_lock`, and user-defined lock-type implementations should provide at least the interface and semantics that is also provided by `unique_lock`.

This section merely presents the interface of the class `condition_variable_any`. As its interface offers the same members as `condition_variable` (allowing, where applicable, passing any lock-type instead of just `unique_lock` to corresponding members), the reader is referred to the previous section for a description of the semantics of the class members.

Like `condition_variable`, the class `condition_variable_any` only offers a default constructor. No copy constructor or overloaded assignment operator is provided.

Also, like `condition_variable`, the class's destructor requires that no thread is blocked by the current thread. This implies that all other (waiting) threads must have been notified; those threads may, however, subsequently block on the lock specified in their `wait` calls.
20.5. EVENT HANDLING (CONDITION VARIABLES)

Note that, in addition to `Lock`, the types `Clock`, `Duration`, `Period`, `Predicate`, and `Rep` are template types, defined just like the identically named types mentioned in the previous section.

Assuming that `MyMutex` is a user defined mutex type, and that `MyLock` is a user defined lock-type (cf. section 20.4 for details about lock-types), then a `condition_variable_any` object can be defined and used like this:

```cpp
MyMutex mut;
MyLock<MyMutex> ul(mut);
condition_variable_any cva;

cva.wait(ul);
```

These are the class `condition_variable_any`’s members:

- `void notify_one()` noexcept;
- `void notify_all()` noexcept;
- `void wait(Lock& lock);`
- `void wait(Lock& lock, Predicate pred);`
- `cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& absTime);`
- `bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& absTime, Predicate pred);`
- `cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& relTime);`
- `bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& relTime, Predicate pred);`

20.5.3 An example using condition variables

Condition variables are used to synchronize threads on the values of data, rather than on the mere access to data (for which plain mutex-objects can be used). Using condition variables, a thread simply sleeps until it is notified by another thread. In a producer-consumer type of program this is usually accomplished like this:

consumer loop:
- wait until there’s an item in store,
  then reduce the number of stored items
- remove the item from the store
- increment the number of available storage locations
- do something with the retrieved item

producer loop:
- produce the next item
- wait until there’s room to store the item,
  then reduce the number of available storage locations
- store the item
- increment the number of stored items
It is important that the two storage administrative tasks (registering the number of available items and available storage locations) are either performed by the client or by the producer. For the consumer ‘waiting’ means:

- Get a lock on the variable containing the actual count
- As long as the count is zero: wait, releasing the lock until another thread has increased the count, then re-acquire the lock.
- Reduce the count
- Release the lock.

This scheme is implemented in a class Semaphore, offering members wait and notify_all. For a more extensive discussion of semaphores see Tanenbaum, A.S. and Austin, T. (2013) Structured Computer Organization, Pearson Prentice-Hall.

The data member containing the actual count is called d_semaphore. It is protected by mutex d_mutex. In addition a condition_variable d_condition is defined:

```cpp
mutable std::mutex d_mutex;
std::condition_variable d_condition;
size_t d_available;
```

The waiting process is implemented through its member function wait:

```cpp
1: void Semaphore::wait()  
2: {  
3:   std::unique_lock<std::mutex> lk(d_mutex); // get the lock
4:   while (d_available == 0)  
5:     d_condition.wait(lk); // internally releases the lock
6:   // and waits, on exit
7:   --d_available; // dec. available
8: } // the lock is released
```

In line 5 d_condition.wait releases the lock. It waits until receiving a notification, and re-acquires the lock just before returning. Consequently, wait’s code always has complete and unique control over d_semaphore.

What about notifying the a waiting thread? This is handled in lines 4 and 5 of the member function notify_all:

```cpp
1: void Semaphore::notify_all()  
2: {  
3:   std::lock_guard<std::mutex> lk(d_mutex); // get the lock
4:   if (d_available++ == 0)  
5:     d_condition.notify_all(); // use notify_one to notify one other
6:   // thread
7: } // the lock is released
```

At line 4 d_semaphore is always incremented; by using a postfix increment it can simultaneously be tested for being zero. If it was initially zero then d_semaphore is now one. A thread waiting until d_semaphore exceeds zero may now continue. A waiting thread is notified by calling
20.6. ATOMIC ACTIONS: MUTEXES NOT REQUIRED

\[ d\text{\_condition}\text{\_.} \text{\_notify\_one}. \text{In situations where multiple threads are waiting 'notify\_all' can also be used.} \]

Using the facilities of the class \textsc{Semaphore} whose constructor expects an initial value of its \texttt{semaphore} data member, the classic consumer-producer paradigm can now be implemented using multi-threading:\footnote{A more elaborate example of the producer-consumer program is found in the \texttt{yo/threading/examples/events.cc} file in the C++ Annotations’s source archive}:

```cpp
Semaphore available(10);
Semaphore filled(0);
std::queue itemQueue;

void consumer()
{
    while (true)
    {
        filled.wait();  // mutex lock the queue with multiple consumers
        size_t item = itemQueue.front();
        itemQueue.pop();
        available.notify_all();
        process(item);     // not implemented here
    }
}

void producer()
{
    size_t item = 0;
    while (true)
    {
        ++item;
        available.wait();  // mutex lock the queue with multiple consumers
        itemQueue.push(item);
        filled.notify_all();
    }
}

int main()
{
    thread consume(consumer);
    thread produce(producer);

    consume.join();
    produce.join();
}
```

### 20.6 Atomic actions: mutexes not required

Before using the facilities introduced in this section the \texttt{<atomic>} header file must be included.

When data are shared among multiple threads, data corruption is usually prevented using mutexes.
To increment a simple `int` using this strategy code as shown below is commonly used:

```cpp
{
    lock_guard<mutex> lk(intVarMutex);
    ++intVar;
}
```

The compound statement is used to limit the `lock_guard`'s lifetime, so that `intVar` is only locked for a short little while.

This scheme is not complex, but at the end of the day having to define a `lock_guard` for every single use of a simple variable, and having to define a matching mutex for each simple variable is a bit annoying and cumbersome.

C++ offers a way out through the use of atomic data types. Atomic data types are available for all basic types, and also for (trivial) user defined types. Trivial types are (see also section 23.6.2) all scalar types, arrays of elements of a trivial type, and classes whose constructors, copy constructors, and destructors all have default implementations, and their non-static data members are themselves of trivial types.

The class template `std::atomic<Type>` is available for all built-in types, including pointer types. E.g., `std::atomic<bool>` defines an atomic bool type. For many types alternative somewhat shorter type names are available. E.g. instead of `std::atomic<unsigned short>` the type `std::atomic_ushort` can be used. Refer to the atomic header file for a complete list of alternate names.

If `Trivial` is a user-defined trivial type then `std::atomic<Trivial>` defines an atomic variant of `Trivial`: such a type does not require a separate mutex to synchronize access by multiple threads.

Objects of the class template `std::atomic<Type>` cannot directly be copied or assigned to each other. However, they can be initialized by values of type `Type`, and values of type `Type` can also directly be assigned to `std::atomic<Type>` objects. Moreover, since `atomic<Type>` types offer conversion operators returning their `Type` values, an `atomic<Type>` objects can also be assigned to or initialized by another `atomic<Type>` object using a `static_cast`:

```cpp
atomic<int> a1 = 5;
atomic<int> a2(static_cast<int>(a1));
```

The class `std::atomic<Type>` provides several public members, shown below. Non-member (free) functions operating on `atomic<Type>` objects are also available.

The `std::memory_order` enumeration defines the following symbolic constants, which are used to specify ordering constraints of atomic operations:

- `memory_order_acq_rel`: the operation must be a read-modify-write operation, combining `memory_order_acquire` and `memory_order_release`;
- `memory_order_acquire`: the operation is an acquire operation. It synchronizes with a release operation that wrote the same memory location;
- `memory_order_consume`: the operation is a consume operation on the involved memory location;
- `memory_order_relaxed`: no ordering constraints are provided by the operation;
- `memory_order_release`: the operation is a release operation. It synchronizes with acquire operations on the same location;
• memory_order_sec_cst: the default memory order specification for all operations. Memory storing operations use memory_order_release, memory load operations use memory_order_acquire, and read-modify-write operations use memory_order_acq_rel.

The memory order cannot be specified for the overloaded operators provided by `atomic<Type>`. Otherwise, most atomic member functions may also be given a final memory_order argument. Where this is not available it is explicitly mentioned at the function’s description.

Here are the standard available `std::atomic<Type>` member functions:

• `bool compare_exchange_strong(Type &currentValue, Type newValue) noexcept:`
  The value in the atomic object is compared to newValue using byte-wise comparisons. If equal true is returned, and newValue is stored in the atomic object; if unequal false is returned and the object's current value is stored in currentValue;

• `bool compare_exchange_weak(Type &oldValue, Type newValue) noexcept:`
  The value in the atomic object is compared to newValue using byte-wise comparisons. If equal true is returned, and newValue is stored in the atomic object; if unequal, or newValue cannot be atomically assigned to the current object false is returned and the object's current value is stored in currentValue;

• `Type exchange(Type newValue) noexcept:`
  The object’s current value is returned, and newValue is assigned to the current object;

• `bool is_lock_free() const noexcept:`
  If the operations on the current object can be performed lock-free true is returned, otherwise false. This member has no memory_order parameter;

• `Type load() const noexcept:`
  The object’s value is returned;

• `operator Type() const noexcept:`
  The object’s value is returned;

• `void store(Type newValue) noexcept:`
  NewValue is assigned to the current object. Note that the standard assignment operator can also be used.

In addition to the above members, integral atomic types ‘Integral’ (essentially the atomic variants of all built-in integral types) also offer the following member functions:

• `Integral fetch_add(Integral value) noexcept:`
  Value is added to the object’s value, and the object’s value at the time of the call is returned;

• `Integral fetch_sub(Integral value) noexcept:`
  Value is subtracted from the object’s value, and the object’s value at the time of the call is returned;
- Integral fetch_and(Integral mask) noexcept:
  The bit-and operator is applied to the object’s value and mask, assigning the resulting value to the current object. The object’s value at the time of the call is returned;

- Integral fetch_|(Integral mask) noexcept:
  The bit-or operator is applied to the object’s value and mask, assigning the resulting value to the current object. The object’s value at the time of the call is returned;

- Integral fetch_^=(Integral mask) noexcept:
  The bit-xor operator is applied to the object’s value and mask, assigning the resulting value to the current object. The object’s value at the time of the call is returned;

- Integral operator++() noexcept:
  The prefix increment operator, returning object’s new value;

- Integral operator++(int) noexcept:
  The postfix increment operator, returning the object’s value before it was incremented;

- Integral operator-() noexcept:
  The prefix decrement operator, returning object’s new value;

- Integral operator-(int) noexcept:
  The postfix decrement operator, returning the object’s value before it was decremented;

- Integral operator+=(Integral value) noexcept:
  Value is added to the object’s current value and the object’s new value is returned;

- Integral operator-=(Integral value) noexcept:
  Value is subtracted from the object’s current value and the object’s new value is returned;

- Integral operator&=(Integral mask) noexcept:
  The bit-and operator is applied to the object’s current value and mask, assigning the resulting value to the current object. The object’s new value is returned;

- Integral operator|=(Integral mask) noexcept:
  The bit-or operator is applied to the object’s current value and mask, assigning the resulting value to the current object. The object’s new value is returned;

- Integral operator^=(Integral mask) noexcept:
  The bit-xor operator is applied to the object’s current value and mask, assigning the resulting value to the current object. The object’s new value is returned;

Some of the free member functions have names ending in _explicit. The _explicit functions define an additional parameter ‘memory_order order’, which is not available for the non-_explicit functions (e.g., atomic_load(atomic<Type> *ptr) and atomic_load_explicit(atomic<Type> *ptr, memory_order order))
20.6. ATOMIC ACTIONS: MUTEXES NOT REQUIRED

Here are the free functions that are available for all atomic types:

- **bool std::atomic_compare_exchange_strong(_explicit)(std::atomic<Type> *ptr, Type *oldValue, Type newValue) noexept:**
  
  returns ptr->compare_exchange_strong(*oldValue, newValue);

- **bool std::atomic_compare_exchange_weak(_explicit)(std::atomic<Type> *ptr, Type *oldValue, Type newValue) noexept:**
  
  returns ptr->compare_exchange_weak(*oldValue, newValue);

- **Type std::atomic_exchange(_explicit)(std::atomic<Type> *ptr, Type newValue) noexept:**
  
  returns ptr->exchange(newValue);

- **void std::atomic_init(std::atomic<Type> *ptr, Type init) noexept:**
  
  Stores init non-atomically in *ptr. The object pointed to by ptr must have been default constructed, and as yet no member functions must have been called for it. This function has no memory_order parameter;

- **bool std::atomic_is_lock_free(std::atomic<Type> const *ptr) noexept:**
  
  returns ptr->is_lock_free(). This function has no memory_order parameter;

- **Type std::atomic_load(_explicit)(std::atomic<Type> *ptr) noexept:**
  
  returns ptr->load();

- **void std::atomic_store(_explicit)(std::atomic<Type> *ptr, Type value) noexept:**
  
  calls ptr->store(value).

In addition to the abovementioned free functions atomic<Integral> types also offer the following free member functions:

- **Integral std::atomic_fetch_add(_explicit)(std::atomic<Integral> *ptr, Integral value) noexcept:**
  
  returns ptr->fetch_add(value);

- **Integral std::atomic_fetch_sub(_explicit)(std::atomic<Integral> *ptr, Integral value) noexcept:**
  
  returns ptr->fetch_sub(value);

- **Integral std::atomic_fetch_and(_explicit)(std::atomic<Integral> *ptr, Integral mask) noexcept:**
  
  returns ptr->fetch_and(value);

- **Integral std::atomic_fetch_or(_explicit)(std::atomic<Integral> *ptr, Integral mask) noexcept:**
  
  returns ptr->fetch_or(value);

- **Integral std::atomic_fetch_xor(_explicit)(std::atomic<Integral> *ptr, Integral mask) noexcept:**
  
  returns ptr->fetch_xor(mask).
CHAPTER 20. MULTI THREADING

20.7 An example: threaded quicksort

The quicksort sorting algorithm (Hoare, 1962) is a well-known sorting algorithm. Given an array of \( n \) elements, it works like this:

- Pick an element from the array, and partition the array with respect to this element (call it the pivot element). This leaves us with two (possibly empty) sub-arrays: one to the left of the pivot element, and one to the right of the pivot element;
- Recursively perform quicksort on the left-hand sub-array;
- Recursively perform quicksort on the right-hand sub-array.

To convert this algorithm to a multi-threaded algorithm appears to be a simple task:

```cpp
void quicksort(Iterator begin, Iterator end)
{
    if (end - begin < 2) // less than 2 elements are left
        return; // and we’re done

    Iter pivot = partition(begin, end); // determine an iterator pointing
                                           // to the pivot element
    thread lhs(quicksort, begin, pivot);// start threads on the left-hand
                                           // side sub-arrays
    thread rhs(quicksort, pivot + 1, end); // and on the right-hand side
                                           // sub-arrays
    lhs.join();
    rhs.join(); // and we’re done
}
```

Unfortunately, this translation to a multi-threaded approach won’t work for reasonably large arrays because of a phenomenon called overpopulation: more threads are started than the operating system is prepared to give us. In those cases a Resource temporarily unavailable exception is thrown, and the program ends.

Overpopulation can be avoided by using a pool of workers, where each ‘worker’ is a thread, which in this case is responsible for handling one (sub) array, but not for the nested calls. The pool of workers is controlled by a scheduler, receiving the requests to sort sub-arrays, and passing these requests on to the next available worker.

The main data structure of the example program developed in this section is a queue of std::pairs containing iterators of the array to be sorted (cf. Figure 20.1, the sources of the program are found in the annotation()’s yo/threading/examples/multisort directory). Two queues are being used: one queue is a task-queue, receiving the iterators of sub-arrays to be partitioned. Instead of immediately launching new threads (the lhs and rhs threads in the above example), the ranges to be sorted are pushed on the task-queue. The other queue is the work-queue: elements are moved from the task-queue to the work-queue, where they will be processed by one of the worker threads.

The program’s main function starts the workforce, reads the data, pushes the arrays begin and end iterators on the task queue and then starts the scheduler. Once the scheduler ends the sorted array is displayed:

```cpp
int main()
```
The workforce consists of a bunch of detached threads. Each thread represents a worker, implemented in the function `void worker`. Since the number of worker threads is fixed, overpopulation doesn’t occur. Once the array has been sorted and the program stops these detached threads simply end:

```c++
for (size_t idx = 0; idx != g_sizeofWorkforce; ++idx)
    thread(worker).detach();
```

The scheduler continues for as long as there are sub-arrays to sort. When this is the case the task queue’s front element is moved to the work queue. This reduces the work queue’s size, and prepares an assignment for the next available worker. The scheduler now waits until a worker is available. Once workers are available one of them is informed of the waiting assignment, and the scheduler waits for the next task:

```c++
void scheduler()
{
    while (newTask())
```
The function `newTask` simply checks whether the task queue is empty. If so, and none of the workers is currently busy sorting a sub-array then the array has been sorted, and `newTask` can return `false`. When the task queue is empty but a worker is still busy, it may be that new sub-array dimensions are going to be placed on the task queue by an active worker. Whenever a worker is active the Semaphore `g_workforce`’s size is less than the size of the work force:

```cpp
bool wip() {
    return g_workforce.size() != g_sizeofWorkforce;
}

bool newTask() {
    bool done;

    unique_lock<mutex> lk(g_taskMutex);
    while ((done = g_taskQ.empty()) && wip())
        g_taskCondition.wait(lk);

    return not done;
}
```

Each detached worker thread performs a continuous loop. In the loop it waits for a notification by the scheduler. Once it receives a notification it retrieves its assignment from the work queue, and partitions the sub-array specified in its assignment. Partitioning may result in new tasks. Once this has been completed the worker has completed its assignment: it increments the available workforce and notifies the scheduler that it should check whether all tasks have been performed:

```cpp
void worker() {
    while (true) {
        g_worker.wait(); // wait for action

        partition(g_workQ.popFront());
        g_workforce.notify_all();

        lock_guard<mutex> lk(g_taskMutex);
        g_taskCondition.notify_one();
    }
}
```

Sub-arrays smaller than two elements need no partitioning. All larger sub-arrays are partitioned relative to their first element. The `std::partition` generic algorithm does this well, but if the
pivot is itself an element of the array to partition then the pivot’s eventual location is undetermined: it may be found anywhere in the series of elements which are at least equal to the pivot. The two required sub-arrays, however, can easily be constructed:

- First call `std::partition` relative to an array’s first element, partitioning the array’s remaining elements, returning `mid`, pointing to the first element of the series of elements that are at least as large as the array’s first element;
- Then swap the array’s first element with element to which `mid - 1` points;
- The two sub-arrays range from, respectively, `array.begin()` to `mid - 1` (elements all smaller than the pivot), and from `mid` to `array.end()` (elements all at least as large as the pivot).

The two iterator pairs defining these two sub-arrays are thereupon added to the task queue, creating two new tasks to be dealt with by the scheduler:

```cpp
void partition(Pair const &range)
{
    if (range.second - range.first < 2)
        return;

    auto rhsBegin = partition(range.first + 1, range.second,
                                bind2nd(less<int>(), *range.first));
    auto lhsEnd = rhsBegin - 1;
    swap(*range.first, *lhsEnd);
    pushTask(range.first, lhsEnd);
    pushTask(rhsBegin, range.second);
}
```

### 20.8 Shared States

Just before a thread ends it may have produced some results. These results may have to be communicated to other threads. In multi threaded programs several classes and functions can be used that produce *shared states*, making it easy to communicate results to other threads. Results could be values, objects or exceptions.

Objects that contain such shared states are called *asynchronous return objects*. However, due to the nature of multi threading, a thread may request the results of an asynchronous return object before these result are actually available. In those cases the requesting thread blocks, waiting for the results to become available. Asynchronous return objects offer `wait` and `get` members which, respectively, `wait` until the results have become available, and `produce` the asynchronous results once they are available. The phrase that is used to indicate that the results are available is ‘the shared state has been made ready’.

Shared states are made ready by *asynchronous providers*. Asynchronous providers are simply objects or functions providing results to shared states. Making a shared state ready means that an asynchronous provider

- marks its shared state as being ready, and
unblocks any waiting threads (e.g., by allowing blocking members, like wait, to return).

Once a shared state has been made ready it contains a value, object, or exception which can be retrieved by objects having access to the shared state. While code is waiting for a shared state to become ready the value or exception that is going to be stored in the shared state may be computed. When multiple threads try to access the same shared state they must use synchronizing mechanisms (like mutexes, cf. section 20.3) to prevent access-conflicts.

Shared states use reference counting to keep track of the number of asynchronous return objects or asynchronous providers that hold references to them. These return objects and providers may release their references to these shared states (which is called 'releasing the shared state). This happens when a return object or provider holds the last reference to the shared state, and the shared state is destroyed.

On the other hand, an asynchronous provider may also abandon its shared state. In that case the provider, in sequence,

- stores an exception object of type std::future_error, holding the error condition std::broken_promise in its shared state;
- makes its shared data ready; and
- releases its shared data.

Objects of the class std::future (see the next section) are asynchronous return objects. They can be produced by the std::async (section 20.11) family of functions, and by objects of the classes std::packaged_task (section 20.12), and std::promise (section 20.13).

20.9 Asynchronous return objects: std::future

Condition variables allow threads to wait until data have obtained certain values. A thread may also have to wait until a sub-thread has finished when calling a sub-thread's join member.

Waiting may be unwelcome: instead of just waiting our thread might also be doing something useful. It might as well pick up the results produced by a sub-thread at some point in the future.

In fact, exchanging data among threads always poses some difficulties, as it requires shared variables, and the use of locks and mutexes to prevent data corruption. Rather than waiting and using locks it would be nice if some asynchronous task could be started, allowing the initiating thread (or even other threads) to pick up the result at some point in the future, when the results are needed, without having to worry about data locks or waiting times. For situations like these C++ provides the class std::future.

Before using the class std::future the <future> header file must be included.

Objects of the class template std::future harbor the results produced by asynchronously executed tasks. The class std::future is a class template. Its template type parameter specifies the type of the result returned by the asynchronously executed task. This type may be void.

On the other hand, the asynchronously executed task may throw an exception (ending the task). In that case the future object catches the exception, and rethrows it once its return value (i.e., the value returned by the asynchronously executed task) is requested.

In this section the members of the class template future are described. Future objects are commonly initialized through anonymous future objects returned by the factory function std::async.
or by the `get_future` members of the classes `std::promise`, and `std::packaged_task` (introduced in upcoming sections). Examples of the use of `std::future` objects are provided in those sections.

Some of `future`'s members return a value of the strongly typed enumeration `std::future_status`. This enumeration defines three symbolic constants: `future_status::ready`, `future_status::timeout`, and `future_status::deferred`.

Error conditions are returned through `std::future_error` exceptions. These error conditions are represented by the values of the strongly typed enumeration `std::future_errc` (covered in the next section).

The class `future` itself provides the following constructors:

- `future()`: The default constructor constructs an `future` object that does not refer to shared results. Its `valid` member returns `false`.
- `future(future &&tmp) noexcept`: The move constructor is available. Its `valid` member returns what `tmp.valid()` would have returned prior to the constructor invocation. After calling the move constructor `tmp.valid()` returns `false`.

The class `future` does not offer a copy constructor or an overloaded assignment operator.

Here are the members of the class `std::future`:

- `future &operator=(future &&tmp)`: The move assignment operator grabs the information from the `tmp` object; following this, `tmp.valid()` returns `false`.
- `std::shared_future<ResultType> share() &&`: Returns a `std::shared_future<ResultType>` (see section 20.10). After calling this function, the `future`’s `valid` member returns `false`.
- `ResultType get()`: First `wait` (see below) is called. Once `wait` has returned the results produced by the associated asynchronous task are returned. With `future<Type>` specifications the returned value is the moved shared value if `Type` supports move assignment, otherwise a copy is returned. With `future<Type &>` specifications a `Type &` is returned, with `future<void>` specifications nothing is returned. If the shared value is an exception, it is thrown instead of returned. After calling this member the `future` object’s `valid` member returns `false`.
- `bool valid() const`: Returns `true` if the `(future)` object for which `valid` is called refers to an object returned by an asynchronous task. If `valid` returns `false`, the `future` object exists, but in addition to `valid` only its destructor and move constructor can safely be called. When other members are called while `valid` returns `false` a `std::future_error` exception is thrown (having the value `future_errc::no_state`).
• **void wait() const:**

The thread is blocked until the results produced by the associated asynchronous task are available.

• **std::future_status wait_for(chrono::duration<Rep, Period> const &rel_time) const:**

This member template derives the template types `Rep` and `Period` from the actually specified duration (cf. section 20.1.2). If the results contain a deferred function nothing happens. Otherwise `wait_for` blocks until the results are available or until the amount of time specified by `rel_time` has expired. Possible return values are:

- `future_status::deferred` if the results contain a deferred function;
- `future_status::ready` if the results are available;
- `future_status::timeout` if the function is returning because the amount of time specified by `rel_time` has expired.

• **future_status wait_until(chrono::time_point<Clock, Duration> const &abs_time) const:**

This member template derives the template types `Clock` and `Duration` from the actually specified `abs_time` (cf. section 20.1.4). If the results contain a deferred function nothing happens. Otherwise `wait_until` blocks until the results are available or until the point in time specified by `abs_time` has expired. Possible return values are:

- `future_status::deferred` if the results contain a deferred function;
- `future_status::ready` if the results are available;
- `future_status::timeout` if the function is returning because the point in time specified by `abs_time` has expired.

The class `std::future<ResultType>` declares the following friends:

```cpp
std::promise<ResultType> (sf. section 20.13), and
```

```cpp
template< typename Function, typename... Args>
std::future<typename result_of<Function(Args...)>::type>
std::async(std::launch, Function &&fun, Args &&...args);
```

(cf. section 20.11).

### 20.9.1 The std::future_error exception and the std::future_errc enum

Members of the class `std::future` may return errors by throwing `std::future_error` exceptions. These error conditions are represented by the values of the strongly typed enumeration `std::future_errc` which defines the following symbolic constants:

• **broken.promise**

`Broken.promise` is thrown when a `future` object was received whose value was never assigned by a `promise` or `packaged_task`. For example, an object of the
20.9. ASYNCHRONOUS RETURN OBJECTS: STD::FUTURE

Class `promise<int>` should set the value of the `future<int>` object returned by its `get_future` member (cf. section 20.13), but if it doesn’t do so, then a `broken_promise` exception is thrown, as illustrated by the following program:

```cpp
1: std::future<int> fun()
2: {
3:     return std::promise<int>().get_future();
4: }
5:
6: int main()
7: try
8: {
9:     fun().get();
10: }
11: catch (std::exception const &exc)
12: {
13:     std::cerr << exc.what() << '\n';
14: }
```

At line 3 a `promise` object is created, but its value is never set. Consequently, it ‘breaks its promise’ to produce a value: when `main` tries to retrieve its value (in line 9) a `std::future_error` exception is thrown containing the the `future_errc::broken_promise` value.

- **future_already_retrieved**

  `Future_already_retrieved` is thrown when multiple attempts are made to retrieve the `future` object from, e.g., a `promise` or `packaged_task` object that (eventually) should be ready. For example:

  ```cpp
  1: int main()
  2: {
  3:     std::promise<int> promise;
  4:     promise.get_future();
  5:     promise.get_future();
  6: }
  ```

  Note that after defining the `std::promise` object in line 3 it has merely been defined: no value is ever assigned to its `future`. Even though no value is assigned to the `future` object, it is a valid object. I.e., after some time the future `should` be ready, and the future’s `get` member should produce a value. Hence, line 4 succeeds, but then, in line 5, the exception is thrown as ‘the future has already been retrieved’.

- **promise_already_satisfied**

  `Promise_already_satisfied` is thrown when multiple attempts are made to assign a value to a `promise` object. Assigning a value or `exception_ptr` to the `future` of a `promise` object may happen only once. For example:

  ```cpp
  1: int main()
  2: {
  3:     std::promise<int> promise;
  4:     promise.set_value(15);
  5:     promise.set_value(155);
  6: }
  ```

- **no_state**

  `No_state` is thrown when a member function (other than `valid`, see below) of a `future` object is called when its `valid` member returns `false`. This happens, e.g.,
when calling members of a default constructed `future` object. No_state is not thrown for `future` objects returned by the async factory function or returned by the `get_future` members of `promise` or `packaged_task` type of objects. Here is an example:

```cpp
1: int main()
2: {
3:   std::future<int> fut;
4:   fut.get();
5: }
```

The class `std::future_error` is derived from the class `std::exception`, and offers, in addition to the `char const *what()` const member also the member `std::error_code const &code()` const, returning an `std::error_code` object associated with the thrown exception.

### 20.10 Shared asynchronous return objects: `std::shared_future`

When a thread activates an asynchronous provider (e.g., a `std::async`) then the return value of the asynchronously called function becomes available in its activating thread through a `std::future` object. The `future` object cannot be used by another thread. If this is required (e.g., see this chapter’s final section) the `future` object must be converted to a `std::shared_future` object.

Before using the class `std::shared_future` the `<future>` header file must be included.

Once a `shared_future` object is available, its `get` member (see below) can repeatedly be called to retrieve the results of the original `future` object. This is illustrated by the next small example:

```cpp
1: int main()
2: {
3:   std::promise<int> promise;
4:   promise.set_value(15);
5:   auto fut = promise.get_future();
6:   auto shared1 = fut.share();
7:   auto shared2 = fut.share();
8:   std::cerr << "Result: " << shared1.get() << 'n';
9:   std::cerr << "Result: " << shared1.get() << 'n';
10:  std::cerr << "Valid: " << fut.valid() << 'n';
11:  std::cerr << "Result: " << shared2.get() << 'n';
12:  std::cerr << "Result: " << shared2.get() << 'n';
13: }
```

In lines 9 and 10 the promise’s results are retrieved multiple times, but having obtained the `shared_future` in line 7, the original `future` object no longer has an associated shared state. Therefore, when another attempt is made (in line 13) to obtain the `shared_future`, a no associated state exception is thrown and the program aborts.

However, multiple copies of `shared_future` objects may co-exist. When multiple copies of
shared_future objects exist (e.g. in different threads), the results of the associated asynchronous task are made ready (become available) at exactly the same moment in time.

The relationship between the classes future and shared_future resembles the relationship between the classes unique_ptr and shared_ptr: there can only be one instance of a unique_pointer, pointing to data, whereas there can be many instances of a shared_pointer, each pointing to the same data.

The effect of calling any member of a shared_future object for which valid() == false other than the destructor, the move-assignment operator, or valid is undefined.

The class shared_future supports the following constructors:

- `shared_future() noexcept`
  
  an empty shared_future object is constructed that does not refer to shared results. After using this constructor the object's valid member returns false.

- `shared_future(shared_future const &other)`
  
  a shared_future object is constructed that refers to the same results as other (if any). After using this constructor the object's valid member returns the same value as other.valid().

- `shared_future(shared_future<Result> &&tmp) noexcept`
  
  Effects: move constructs a shared_future object that refers to the results that were originally referred to by tmp (if any). After using this constructor the object's valid member returns the same value as tmp.valid() returned prior to the constructor invocation, and tmp.valid() returns false.

- `shared_future(future<Result> &&tmp) noexcept`
  
  Effects: move constructs a shared_future object that refers to the results that were originally referred to by tmp (if any). After using this constructor the object's valid member returns the same value as tmp.valid() returned prior to the constructor invocation, and tmp.valid() returns false.

The class's destructor destroys the shared_future object for which it is called. If the object for which the destructor is called is the last shared_future object, and no std::promise or std::packaged_task is associated with the results associated with the current object, then the results are also destroyed.

Here are the members of the class std::shared_future:

- `shared_future& operator=(shared_future &&tmp)`:
  
  The move assignment operator releases the current object’s shared results, and move assigns tmp’s results to the current object. After calling the move assignment operator the current object’s valid member returns the same value as tmp.valid() returned prior to the invocation of the move assignment operator, and tmp.valid() returns false;

- `shared_future& operator=(shared_future const &rhs)`:
  
  The assignment operator releases the current object’s shared results, and rhs’s results are shared with the current object. After calling the assignment operator the current object’s valid member returns the same value as tmp.valid();
• Result const &shared_future::get() const:

(Specializations for shared_future<Result &> and shared_future<void> are also available). This member waits until the shared results are available, and subsequently returns Result const &. Note that access to the data stored in Results, accessed through get is not synchronized. It is the responsibility of the programmer to avoid race conditions when accessing Result’s data. If Result holds an exception, it is thrown when get is called;

• bool valid() const:

Returns true if the current object refers to shared results;

• void wait() const:

Blocks until shared results are available (i.e., the associated asynchronous task has produced results);

• future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const:

(The template types Rep and Period normally are derived by the compiler from the actual rel_time specification.) If the shared results contain a deferred function (cf. section 20.11) nothing happens. Otherwise wait_for blocks until the results of the associated asynchronous task has produced results, or until the relative time specified by rel_time has expired. The member returns

- future_status::deferred if the shared results contain a deferred function;
- future_status::ready if the shared results are available;
- future_status::timeout if the function is returning because the amount of time specified by rel_time has expired;

• future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const:

(The template types Clock and Duration normally are derived by the compiler from the actual abs_time specification.) If the shared results contain a deferred function nothing happens. Otherwise wait_until blocks until the shared results are available or until the point in time specified by abs_time has expired. Possible return values are:

- future_status::deferred if the shared results contain a deferred function;
- future_status::ready if the shared results are available;
- future_status::timeout if the function is returning because the point in time specified by abs_time has expired.

### 20.11 Starting a new thread: std::async

In this section the function template std::async is covered. Async is used to start asynchronous tasks, returning values (or void) to the calling thread, which is hard to realize merely using the std::thread class.

Before using the function async the <future> header file must be included.

When starting a thread using the facilities of the class std::thread the initiating thread at some point commonly calls the thread's join method. At that point the thread must have finished or execution blocks until join returns. While this often is a sensible course of action, it may not
always be: maybe the function implementing the thread has a return value, or it could throw an exception.

In those cases join cannot be used: if an exception leaves a thread, then your program ends. Here is an example:

```cpp
void thrower()
{
    throw std::exception();
}

int main()
try
{
    std::thread subThread(thrower);
}
catch (...) {
{
    std::cerr << "Caught exception\n";
}
In line 3 thrower throws an exception, leaving the thread. This exception is not caught by main’s try-block (as it is defined in another thread). As a consequence, the program terminates.

This scenario doesn’t occur when std::async is used. Async may start a new asynchronous task, and the activating thread may retrieve the return value of the function implementing the asynchronous task or any exception leaving that function from a std::future object returned by the async function. Basically, async is called similarly to the way a thread is started using std::thread: it is passed a function and optionally arguments which are forwarded to the function.

Although the function implementing the asynchronous task may be passed as first argument, async first argument may also be a value of the strongly typed enumeration std::launch:

```cpp
enum class launch
{
    async,
    deferred
};
```

When passing launch::async the asynchronous task immediately starts; when passing launch::deferred the asynchronous task is deferred. When std::launch is not specified the default value launch::async |launch::deferred is used, giving the implementation freedom of choice, usually resulting in deferring execution of the asynchronous task.

So, here is the first example again, this time using async to start the sub-thread:

```cpp
1: bool fun()
2: {
3:     return std::cerr << " hello from fun\n";
4: }
5: int exceptionalFun()
6: {
7:     throw std::exception();
```
```cpp
8: }
9: 
10: int main()
11: try
12: {
13:   auto fut1 = std::async(std::launch::async, fun);
14:   auto fut2 = std::async(std::launch::async, exceptionalFun);
15:   std::cerr << "fun returned " << std::boolalpha << fut1.get() << 'n';
16:   std::cerr << "exceptionalFun did not return " << fut2.get() << 'n';
17: }
18: catch (...) {
19:   std::cerr << "caught exception thrown by exceptionalFun\n";
20: }
```

Now the threads immediately start, but although the results are available around line 13, the thrown exception isn't terminating the program. The first thread's return value is made available in line 16, the exception thrown by the second thread is simply caught by main's try-block (line 19).

The function template `async` has several overloaded versions:

- The basic form expects a function or functor as its first argument, returning a `std::future` holding the function's return value or exception thrown by the function:
  ```cpp
template <typename Function, class ...Args>
std::future<typename std::result_of<Function(Args ...)>::type>
  std::async(Function &&fun, Args &&...args);
```

- Alternatively, the first argument may be the address of a member function. In that case the (required) second argument is an object (or a pointer to an object) of that member function's class. Any remaining arguments are passed to the member function (see also the remarks below).

- The first argument may also be a combination (using the `bit_or` operator) of the enumeration values of the `std::launch` enumeration:
  ```cpp
template <class Function, class ...Args>
std::future<typename std::result_of<Function(Args ...)>::type>
  std::async(std::launch policy, Function &&fun, Args &&...args);
```

- If the first argument specifies `std::launch` values, the second argument may also be the address of a member function. In that case the (required) thirs argument is an object (or a pointer to an object) of that member function's class. Any remaining arguments are passed to the member function (see also the remarks below).

When calling `async` all arguments except for the `std::launch` argument must be references, pointers or move-constructible objects:

- When a member function is specified, then the object for which the member function is called must be a named object, an anonymous object, or a pointer to a named object.

- When a named object is passed to the `async` function template then copy construction is used to construct a copy of the argument which is then forwarded to the thread-launcher.
• When an anonymous object is passed to the `async` function template then move construction is used to forward the anonymous object to the thread launcher.

Once the thread itself starts another move construction is used to construct an object for the duration of the thread. When a pointer to an object is passed, the sub-thread uses the object referred to by the pointer, and neither copy- nor move-construction is required. However, when using a pointer to an object the programmer should make sure that the object’s lifetime exceeds the duration of the thread (note that this is not automatically guaranteed, as the asynchronous task may not actually start before the future’s `get` member is called).

Because of the default `std::launch::deferred | std::launch::async` argument used by the basic `async` call it is likely that the function which is passed to `async` doesn’t immediately start. The `launch::deferred` policy allows the implementor to defer its execution until the program explicitly asks for the function’s results. Consider the following program:

```cpp
#include <iostream>
#include <future>

//code
bool fun()
{
    return std::cerr << " hello from fun\n";
}

int exceptionalFun()
{
    throw std::exception();
}

int main()
try
{
    auto fut1 = std::async(std::launch::async, fun);
    auto fut2 = std::async(std::launch::async, exceptionalFun);

    std::cerr << "fun returned " << std::boolalpha << fut1.get() << 'n';
    std::cerr << "exceptionalFun did not return " << fut2.get() << 'n';
}
catch (...)
{
    std::cerr << "caught exception thrown by exceptionalFun\n";
}
//=
```

Although `async` is called in line 9, when the program is run its output may not show `fun`’s output line. This is a result of the use of `launch::deferred`: the system simply defers `fun`’s execution until requested, which doesn’t happen. But the future object that’s returned by `async` has a member `wait`. Once `wait` returns the shred state must be available. In other words: `fun` must have finished. Here is what happens when line 14 is changed into `async(fun).wait()`:

First async call starts
hello from fun
First async call ends
Second async call starts
hello from fun
Evaluation of `fun` can thus be controlled to the point where we want it, maybe even after calling `asyncCall`, as shown in the next example.

The `std::async` function template is used to start a thread, making its results available to the calling thread. On the other hand, we may only be able to prepare (package) a task (a thread), but may have to leave the completion of the task to another thread. Scenarios like this are realized through objects of the class `std::package_task`, which is the topic of the next section.

### 20.12 Preparing a task for execution: `std::packaged_task`

The class template `std::packaged_task` allows a program to ‘package’ a function or functor and pass the package to a thread for further processing. The processing thread then calls the packaged function, passing it its arguments (if any). After completing the function the `packaged_task`’s future is ready, allowing the program to retrieve the results produced by the function. Thus, functions and the results of function calls can be transferred between threads.

Before using the class template `packaged_task` the `<future>` header file must be included.

Before describing the class’s interface, let’s first look at an example to get an idea about how a `packaged_task` can be used. Remember that the essence of `packaged_task` is that part of your program prepares (packages) a task for another thread to complete, and that the program at some point needs the result of the completed task.

To clarify what’s happening here, let’s first look at a real-life analogon. Every now and then I make an appointment with my garage to have my car serviced. The ‘package’ in this case are the details about my car: its make and type determine the kind of actions my garage performs when servicing it. My neighbor also has a car, which also needs to be serviced every now and then. This also results in a ‘package’ for the garage. At the appropriate time me and my neighbor take our cars to the garage (i.e., the packages are passed to another thread). The garage services the cars (i.e., calls the functions stored in the `packaged_tasks` [note that the tasks differ, depending on the types of the cars]), and performs some actions that are associated with it (e.g., registering that my or my neighbor’s car has been serviced, or order replacement parts). In the meantime my neighbor and I perform our own businesses (the program continues while a separate thread runs as well). But by the end of the day we’d like to use our cars again (i.e., get the results associated with the `packaged_task`). A common result in this example is the garage’s bill, which we have to pay (the program obtains the `packaged_task`’s results).

Here is a little C++ program illustrating the use of a `packaged_task` (assuming the required headers and `using namespace std` have been specified):

```cpp
1: mutex carDetailsMutex;
2: condition_variable condition;
3: string carDetails;
4: packaged_task<size_t (std::string const &)> serviceTask;
5:
6: size_t volkswagen(string const &type)
7: {
8:     cout << "performing maintenance by the book for a " << type << ' \n';
9:     return type.size() * 75; // the size of the bill
10: }
11:
```
12: size_t peugeot(string const &type)
13: {
14:     cout << "performing quick and dirty maintenance for a " << type << 'n';
15:     return type.size() * 50; // the size of the bill
16: }
17:
18: void garage()
19: {
20:     while (true)
21:     {
22:         unique_lock<mutex> lk(carDetailsMutex);
23:         while (carDetails.empty())
24:             condition.wait(lk);
25:         
26:         cout << "servicing a " << carDetails << 'n';
27:         serviceTask(carDetails);
28:         carDetails.clear();
29:     }
30: }
31:
32: int main()
33: {
34:     thread(garage).detach();
35:     
36:     while (true)
37:     {
38:         string car;
39:         if (not getline(cin, car) || car.empty())
40:             break;
41:         {
42:             lock_guard<mutex> lk(carDetailsMutex);
43:             carDetails = car;
44:         }
45:         serviceTask = packaged_task<size_t (string const &)> (car[0] == 'v' ? volkswagen : peugeot
46:             car[0] == 'v' ? volkswagen : peugeot
47:         );
48:         auto bill = serviceTask.get_future();
49:         condition.notify_one();
50:         cout << "Bill for servicing a " << car << ": EUR " << bill.get() << 'n';
51:     }
52: }
53: }

- Lines 1-3 define the variables used for synchronization;
- Line 4 defines a packaged_task: serviceTask is initialized with a function (or functor) expecting a string, returning a size_t;
- Lines 6-10 and 12-16 define such functions: volkswagen and peugeot represent the tasks to perform when cars of the provided types come in for service; presumably they return the bill.
- Lines 18-30 define the function void garage, defining the actions performed by the garage when cars come in for service. These actions are performed by a separate detached thread, starting in line 34. In a continuous loop it waits until it obtains a lock on the carDetailsMutex and carDetails is no longer empty. Then, at line 27, it passes
carDetails to the packaged_task ‘serviceTask’. By itself this is not identical to calling the packaged_task’s function, but eventually its function will be called. At this point the packaged_task receives its function’s arguments, which it eventually will forward to its configured function. Finally, at line 28 it clears carDetails, thus preparing itself for the next request.

• Lines 32-53 define main:
  – First, at line 34 the anonymous detached thread running garage is started.

Then the program’s main loop starts (lines 36-52):
  – The main thread reads commands from the standard input until an empty or no line is received (lines 38-40).
  – By convention the line’s first letter starts the car’s brand (volkswagen or peugeot), and the packaged_task, provided with the right servicing function, is constructed next (line 45).
  – Then, at line 48 the results, stored in a future, are retrieved. Although at this point the future might not be ready, the future object itself is, and it is simply returned as the bill.
  – Now we’re ready to inform the garage that it can service a car: the garage is notified in line 49.

Anything may happen next: the program may perform any actions, but eventually it requests the results produced by the garage.
  – The main thread obtains the results by calling bill.get() in line 51. If, by this time, the car is still being serviced, the bill isn’t ready yet, and bill.get() blocks until it is, and the bill for servicing a car is shown.

Now that we’ve seen an example of a program using a packaged_task, let’s have a look at its interface. Note that the class packaged_task is a class template: its template type parameter specifies the prototype of a function or function object implementing the task performed by the packaged_task object.

Constructors and destructor:

• packaged_task() noexcept:
  The default constructor constructs a packaged_task object which is not associated with a function or shared state;

• explicit packaged_task(ReturnType(Args...) &&function):
  A packaged_task is constructed for a function or functor expecting arguments of types Args..., and returning a value of type ReturnType. The packaged_task class template specifies ReturnType (Args...) as its template type parameter. The constructed object contains a shared state, and a (move constructed) copy of function.
  Optionally an Allocator may be specified as second template type parameter, in which case the first two arguments are std::allocator_arg_t, Allocator const &alloc. The type std::allocator_arg_t is a type introduced to disambiguate constructor selections, and can simply be specified as std::allocator_arg_t().
  This constructor may throw a std::bad_alloc exception or exceptions thrown by function’s copy or move constructors
• packaged_task(packaged_task &&tmp) noexcept:

The move constructor moves any existing shared state from tmp to the newly constructed object, removing the shared state from tmp.

• ~packaged_task():

The object’s shared state (if any) is abandoned

Member functions:

• future<ReturnType> get_future():

A std::future object haring the current object’s shared state is returned. A future_error exception is thrown upon error, containing

- future_already_retrieved if get_future was already called on a packaged_task object containing the same shared state as the current object;
- no_state if the current object has no shared state.

Note: Any futures that share the object’s shared state may access the result returned by the object’s task.

• void make_ready_at_thread_exit(Args... args):

Calls void operator()(Args... args) (see below) when the current thread exits, once all objects of thread storage duration associated with the current thread have been destroyed.

• packaged_task &operator=(packaged_task &&tmp):

The move assignment operator first releases the current object’s shared state (if available), after which the current object and tmp are swapped;

• void operator()(Args... args):

The args arguments are forwarded to the current object’s stored task. When the stored task returns its return value is stored in the current object’s shared state. Otherwise any exception thrown by the task is stored in the object’s shared state. Following this the object’s shared state is made ready, and any threads blocked in a function waiting for the object’s shared state to become ready are unblocked. A future_error exception is thrown upon error, containing

- promise_already_satisfied if the shared state has already been made ready;
- no_state if the current object does not have any shared state.

Calling this member synchronizes with calling any member function of a (shared_)future object that provides access to the packaged_task’s results.

• void reset():

Abandons any available shared state, initializing the current object to packaged_task(std::move(funct)), where funct is the object’s stored task. This member may throw the following exceptions:

- bad_alloc if memory for the new shared state could not be allocated;
- any exception thrown by the move constructor of the task stored in the shared state;
- future_error with a no_state error condition if the current object contains no shared state.
• void swap(packaged_task &other) noexcept:
  The shared states and stored tasks of the current object and other are swapped.

• bool valid() const noexcept:
  Returns true if the current object contains a shared state, otherwise false is returned;

The following non-member (free) function operating on packaged_task objects is available:

• void swap(packaged_task<ReturnType(Args...)>& lhs, packaged_task<ReturnType(Args...)>& rhs) noexcept
  Calls lhs.swap(rhs)

20.13 The class ‘std::promise’

In addition to std::packaged_task and std::async the class template std::promise can be used to obtain the results from a separate thread.

Before using the class template promise the <future> header file must be included.

A promise is useful to obtain the results from another thread without further synchronization requirements. Consider the following program:

```cpp
void compute(int *ret)
{
    *ret = 9;
}

int main()
{
    int ret = 0;
    std::thread(compute, &ret).detach();
    cout << ret << endl;
}
```

Chances are that this program shows the value 0: the cout statement is already executed before the detached thread has had a chance to complete its work. In this example that problem can easily be solved by using a non-detached thread, and using the thread’s join member, but when multiple threads are used that requires named threads and as many join calls. Instead, using a promise might be preferred:

```cpp
1: void compute1(promise<int> &ref)  
2: {  
3:     ref.set_value(9);  
4: }  
5:  
6: int main()  
7: {  
8:     std::promise<int> p;  
9:     std::thread(compute, ref(p)).detach();  
```
10:     cout << p.get_future().get() << ‘\n’;
12: }

In this example again a detached thread is used, but its results are kept for future reference in a promise object, instead of directly being assigned to a final destination variable. The promise object contains a future object holding the computed value. The future’s get member blocks until the future has been made ready, at which point the result becomes available. By then the detached thread may or may not yet have been completed. If it already completed its work then get immediately returns, otherwise there will be a slight delay.

Promises can be useful when implementing a multi threaded version of some algorithm without having to use additional synchronization statements. As an example consider matrix multiplications. Each element of the resulting product matrix is computed as the inner product of two vectors: the inner product of a row of the left-hand matrix operand and a column of the right-hand matrix operand becomes element [row][column] of the resulting matrix. Since each element of the resulting matrix can independently be computed from the other elements, a multi threaded implementation is well possible. In the following example the function innerProduct (lines 4..11) leaves its result in a promise object:

```
1: int m1[2][2] = {{1, 2}, {3, 4}};
2: int m2[2][2] = {{3, 4}, {5, 6}};
3:
4: void innerProduct(promise<int> &ref, int row, int col)
5: {
6:     int sum = 0;
7:     for (int idx = 0; idx != 2; ++idx)
8:         sum += m1[row][idx] * m2[idx][col];
9:     ref.set_value(sum);
11: }
12:
13: int main()
14: {
15:     promise<int> result[2][2];
16:
17:     for (int row = 0; row != 2; ++row)
18:         {
19:         for (int col = 0; col != 2; ++col)
20:             thread(innerProduct, ref(result[row][col]), row, col).detach();
21:         }
22:
23:     for (int row = 0; row != 2; ++row)
24:         {
25:         for (int col = 0; col != 2; ++col)
26:             cout << setw(3) << result[row][col].get_future().get();
27:         cout << ‘\n’;
28:         }
29: }
```

Each inner product is computed by a separate (anonymous and detached) thread (lines 17..21), which starts as soon as the run-time system allows it to start. By the time the threads have finished the resulting inner products can be retrieved from the promises’ futures. Since futures’ get members block until their results are actually available, the resulting matrix can simply be displayed by
calling those members in sequence (lines 23..28).

So, a promise allows us to use a thread to compute a value (or exception, see below), which value may then be collected by another thread at some future point in time. The promise remains available, and as a consequence further synchronization of the threads and the program starting the threads is not necessary. When the promise object contains an exception, rather than a value, its future’s get member rethrows the stored exception.

Here is the class promise’s interface. Note that the class promise is a class template: its template type parameter ReturnType specifies the template type parameter of the std::future that can be retrieved from it.

Constructors and destructor:

- **promise()**:

  The default constructor constructs a promise object containing a shared state. The shared state may be returned by the member get_future (see below), but that future has not yet been made ready;

- **promise(promise &&tmp) noexcept**:

  The move constructor constructs a promise object, transferring the ownership of tmp’s shared state to the newly constructed object. After the object has been constructed, tmp no longer contains a shared state;

- **~promise()**:

  The object’s shared state (if any) is abandoned;

Member functions:

- **std::future<ReturnType> get_future()**:

  A std::future object sharing the current object’s shared state is returned. A future_error exception is thrown upon error, containing

    - future_already_retrieved if get_future was already called on a packaged_task object containing the same shared state as the current object;
    - no_state if the current object has no shared state.

  Note: Any futures that share the object’s shared state may access the result returned by the object’s task;

- **promise &operator=(promise &&rhs) noexcept**:

  The move assignment operator first releases the current object’s shared state (if available), after which the current object and tmp are swapped;

- **void promise<void>::set_value()**:

  See below, at the last set_value member’s description;

- **void set_value(ReturnType &&value)**:

  See below, at the last set_value member’s description;

- **void set_value(ReturnType const &value)**:

  See the next member function’s description;
• void set_value(ReturnType &value):
  The argument (value) is atomically stored in the shared state, which is then also made ready. A future_error exception is thrown upon error, containing
  – promise_already_satisfied if the shared state has already been made ready;
  – no_state if the current object does not have any shared state.
  Alternatively, any exception thrown by value’s move or copy constructor may be thrown;

• void set_exception(std::exception_ptr obj):
  Exception_ptr obj (cf. section 20.13.1) is atomically stored in the shared state, making that state ready. A future_error exception is thrown upon error, containing
  – promise_already_satisfied if the shared state has already been made ready;
  – no_state if the current object does not have any shared state;

• void set_exception_at_thread_exit(exception_ptr ptr):
  The exception pointer ptr is stored in the shared state without immediately making that state ready. The state becomes ready when the current thread exits, once all objects of thread storage duration which are associated with the ending thread have been destroyed. A future_error exception is thrown upon error, containing
  – promise_already_satisfied if the shared state has already been made ready;
  – no_state if the current object does not have any shared state;

• void swap(promise& other) noexcept:
  The shared states (if any) of the current object and other are exchanged.

The following non-member (free) function operating on promise objects is available:

• void swap(promise<ReturnType> &lhs, promise<ReturnType> &rhs) noexcept:
  Calls lhs.swap(rhs)
20.13.1 Exception propagation: std::exception_ptr

Std::promise’s member set_exception does not expect an std::exception argument, but an object of the class std::exception_ptr. In this section we take a closer look at the class exception_ptr.

Before using the class exception_ptr the <future> header file must be included.

The class an exception_ptr’s default constructor initializes it to a null-pointer. In the following code snippet the variable isNull is set to true:

```cpp
std::exception_ptr obj;
bool isNull = obj == nullptr && obj == 0;
```

The class exception_ptr provides copy and move constructors as well as copy and move assignment operators.

Two exception_ptr objects can be compared for equality. They are equal if they refer to the same exception. Move assignment transfers the exception referred to by the right-hand side operand to the left-hand side operand, and turns the right-hand side operand into a null pointer.

There is no published method directly retrieving the exception to which an exception_ptr object refers. However, there are some free functions constructing or handling exception_ptr objects:

- **std::exception_ptr std::current_exception() noexcept:**
  
  An exception_ptr object is returned referring to the currently handled exception (or a copy of the currently handled exception, or a default constructed exception_ptr object if no current exception is available). This function can also be called when a default exception catcher is used. E.g., assuming that obj refers to an available std::promise object, then the following code snippet assigns the exception caught by default catch clause to obj:

  ```cpp
  ...  
  catch (...)  
  {  
      obj.set_exception(std::current_exception());  
  }
  ```

  The exception referred to by current_exception does not have to be an object of the class std::exception. Any type of object or value thrown as an exception is retrieved as an exception_ptr by current_exception. The exception referred to by an exception_ptr object remains valid for at least as long as there exists an exception_ptr object that refers to it. Calling current_exception twice in a row then the two returned exception_ptr objects may or may not refer to the same exception object.

- **std::exception_ptr make_exception_ptr(Type value) noexcept:**

  This function template constructs an exception_ptr from a value of any type which is passed as its argument. Type does not necessarily have to be an std::exception. The constructed exception_ptr could, e.g., be assigned to a std::promise. When the promise’s future’s get member is subsequently called (possibly from within another thread) the exception will be thrown. Here are some examples, showing how values of different types can be passed as arguments to make_exception_ptr, and
20.14. AN EXAMPLE: MULTI-THREADED COMPILATIONS

showing how the eventually constructed exception_ptr is assigned to the obj, which is assumed to be of a std::promise type:

```cpp
auto ptr = make_exception_ptr(exception());
ptr = make_exception(string("hello world"));
ptr = make_exception(12);
```

- void std::rethrow_exception(exception_ptr obj):

  The exception to which obj refers is thrown. Note: obj cannot be a nullptr.

20.14 An example: multi-threaded compilations

In this section another program is developed. This section’s example program illustrates the use of packaged_tasks.

Like the multi-threaded quicksort example a worker pool is used. However, in this example the workers in fact do not know what their task is. In the current example the tasks happens to be identical, but different tasks might as well have been used, without the need to update the workers.

The program uses a class Task containing a command-specification (d_command, and a task specification (d_task) (cf. Figure 20.2, the sources of the program are found in the yo/threading/examples/multicompile directory of the C++ Annotations).

Like before main first starts its workforce as detached threads. Following this, the compilation jobs are performed by yet another detached thread. Eventually, the results of the compilation jobs are handled by results:

```cpp
int main()
{
    workforce(); // start the worker threads
    thread(jobs).detach(); // start working on the jobs
    results(); // handle the results.
}
```

The jobs-thread reads the names of the files to compile from the standard input stream, and passes them on to dispatch for further handling by the workers. Lines 7 and 8 are executed at the end, and are a safeguard against empty input. The safeguard is discussed below, at newResult’s description.

```cpp
1: void jobs()
2: {
3:     string line;
4:     while (getline(cin, line) && dispatch(line))
5:         ;
6:     g_ready = true;
7:     g_resultCond.notify_one();
8: }
Next the dispatcher. It ignores empty lines. Also, if a compilation has failed by the time the dispatcher is called, processing stops (lines 6-7). Otherwise, the dispatcher waits for an available worker, prepares a new task, and notifies a worker to handle it:

```cpp
bool dispatch(string const &line)
{
    if (line.empty())
        return true;

    if (g_done.load())
        return false;

    g_workforce.wait();
    newTask(line);
    g_worker.notify_all();
    return true;
}
```

The function `newTask` prepares the program for the next task. First a `Task` object is created. `Task` contains the name of the file to compile, and a `packaged_task`. It encapsulates all activities that are associated with a `packaged_task`. Here is its (in-class) definition:

```cpp
1: typedef packaged_task<Result (string const &fname)> PackagedTask;
2: class Task
3: 
```
4: {
5:    string d_command;
6:    PackagedTask d_task;
7:    
8:    public:
9:        Task() = default;
10:       
11:        Task(string const &command, PackagedTask &&tmp)
12:           :
13:            d_command(command),
14:            d_task(move(tmp))
15:           {}
16: 
17:        void operator()()
18:           {
19:            d_task(d_command);
20:           }
21: 
22:        shared_future<Result> result()
23:           {
24:            return d_task.get_future().share();
25:           }
26:    };

Note (lines 22-25) that result returns a shared_future. Since the dispatcher runs in a different thread than the one processing the results, the futures created by the dispatcher must be shared with the futures required by the function processing the results. Hence the shared_futures returned by Task::result.

Once a Task object has been constructed its shared_future is pushed on the result queue. Although the actual results aren’t available by this time, the result function (see below) is notified that something has been pushed on its queue. Additionally, the Task itself is pushed on the task queue, where a worker may retrieve it:

typedef packaged_task<Result (string const &fname)> PackagedTask;

class Task
{
    string d_command;
    PackagedTask d_task;

    public:
    Task() = default;

    Task(string const &command, PackagedTask &&tmp)
    :
        d_command(command),
        d_task(move(tmp))
    {}

    void operator()()
    {
        d_task(d_command);
    }
shared_future<Result> result()
{
    return d_task.get_future().share();
}

void pushResultQ(shared_future<Result> const &sharedResult)
{
    lock_guard<mutex> lk(g_resultQMutex);
    g_resultQ.push(sharedResult);
    g_ready = true;
    g_resultCond.notify_one();
}

The workers have a simple task: retrieve the next task from the task queue's front, then perform that task. Whatever happens inside the tasks themselves is of no concern: the worker thread eventually ends when the program ends:

void worker()
{
    Task task;

    while (true)
    {
        g_worker.wait();

        g_taskQ.popFront(task);
        task();

        g_workforce.notify_all();
    }
}

This completes the description of how tasks are handled. The task itself remains to be described. In the current example program source files are being compiled. The compilation command is passed to the constructor of a CmdFork object, which starts the compiler as a child process. The result of the compilation is retrieved via its childExit (returning the compiler's exit code) and childOutput (returning any textual output produced by the compiler) members. If compilation fails, the exit value won't be zero. In this case no further compilation tasks will be initiated (lines 11 and 12; the implementation of the class CmdFork is available from the C++ Annotations' yo/threading/examples/cmdfork directory). Here is the function compile:

1: Result compile(string const &line)
2: {
3:     string command("/usr/bin/g++ -Wall --std=c++11 -c " + line);
4:     CmdFork cmdFork(command);
5:     cmdFork.fork();
6:     Result ret {cmdFork.childExit() == 0,
7:         line + "\n" + cmdFork.childOutput()};
8: }
9: 
10: }
11: if (not ret.ok)
12:     g_done = true;
13: 
14: return ret;
15: }

The `results` function continues for as long as there are results. Once results are available they are displayed. Whenever a compilation has failed the `Result`'s `d_ok` member is false and `results` ends:

```cpp
void results()
{
    while (newResult())
    {
        Result const &result = g_resultQ.front().get();
        cerr << result.display;
        if (not result.ok)
            return;
        g_resultQ.pop();
    }
}
```

The function `newResult` controls `results`' while-loop. It returns `true` when there are some results available in the result queue. When no filenames are presented at the standard input stream or when no result has as yet been pushed on the result queue `newResult` waits. It also waits when there are no results available in the results queue, but at least one worker thread is busy. Whenever a result is pushed on the result queue, and also once the input stream has been processed `newResult` is notified. Following the notification `newResult` returns, and `results` either ends or shows the results appearing at the result queue's front:

```cpp
bool newResult()
{
    unique_lock<mutex> lk(g_resultQMutex);

    while
    {
        if (g_resultQ.empty()
            ||
                g_workforce.size() != g_sizeOfWorkforce
            ||
                g_ready.load())
        {
            g_resultCond.wait(lk);
            return not g_resultQ.empty();
        }
```
Chapter 21

Function Templates

C++ supports syntactic constructs allowing programmers to define and use completely general (or abstract) functions or classes, based on generic types and/or (possibly inferred) constant values. In the chapters on abstract containers (chapter 12) and the STL (chapter 18) we’ve already used these constructs, commonly known as the template mechanism.

The template mechanism allows us to specify classes and algorithms, fairly independently of the actual types for which the templates are eventually going to be used. Whenever the template is used, the compiler generates code that is tailored to the particular data type(s) used with the template. This code is generated at compile-time from the template’s definition. The piece of generated code is called an instantiation of the template.

In this chapter the syntactic peculiarities of templates are covered. The notions of template type parameter, template non-type parameter, and function template are introduced and several examples of templates are provided (both in this chapter and in chapter 24). Template classes are covered in chapter 22.

Templates already offered by the language include the abstract containers (cf. chapter 12); the string (cf. chapter 5); streams (cf. chapter 6); and the generic algorithms (cf. chapter 19). So, templates play a central role in present-day C++, and should not be considered an esoteric feature of the language.

Templates should be approached somewhat similarly as generic algorithms: they’re a way of life; a C++ software engineer should actively look for opportunities to use them. Initially, templates may appear to be rather complex and you might be tempted to turn your back on them. However, over time their strengths and benefits are more and more appreciated. Eventually you’ll be able to recognize opportunities for using templates. That’s the time where your efforts should no longer focus on constructing ordinary functions and classes (i.e., functions or classes that are not templates), but on constructing templates.

This chapter starts by introducing function templates. The emphasis is on the required syntax. This chapter lays the foundation upon which the other chapters about templates are built.

21.1 Defining function templates

A function template’s definition is very similar to the definition of a normal function. A function template has a function head, a function body, a return type, possibly overloaded definitions, etc.. However, different from ordinary functions, function templates always use one or more formal types:
types for which almost any existing (class or primitive) type could be used. Let’s have a look at a simple example. The following function \texttt{add} expects two \texttt{Type} arguments and returns their sum:

\begin{verbatim}
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}
\end{verbatim}

Note how closely the above function’s definition follows its description. It receives two arguments, and returns its sum. Now consider what would happen if we defined this function for, e.g., \texttt{int} values. We would write:

\begin{verbatim}
int add(int const &lvalue, int const &rvalue)
{
    return lvalue + rvalue;
}
\end{verbatim}

So far, so good. However, were we to add two doubles, we would overload this function:

\begin{verbatim}
double add(double const &lvalue, double const &rvalue)
{
    return lvalue + rvalue;
}
\end{verbatim}

There is no end to the number of overloaded versions we might be forced to construct: an overloaded version for \texttt{string}, for \texttt{size_t}, for \ldots. In general, we would need an overloaded version for every type supporting \texttt{operator+} and a copy constructor. All these overloaded versions of basically the same function are required because of the strongly typed nature of \texttt{C++}. Because of this, a truly generic function cannot be constructed without resorting to the template mechanism.

Fortunately, we’ve already seen an important part of a template function. Our initial function \texttt{add} actually is an implementation of such a function although it isn’t a full template definition yet. If we gave the first \texttt{add} function to the compiler, it would produce an error message like:

\begin{verbatim}
error: ‘Type’ was not declared in this scope
error: parse error before ‘const’
\end{verbatim}

And rightly so, as we failed to define \texttt{Type}. The error is prevented when we change \texttt{add} into a full template definition. To do this, we look at the function’s implementation and decide that \texttt{Type} is actually a \emph{formal} typename. Comparing it to the alternate implementations, it is clear that we could have changed \texttt{Type} into \texttt{int} to get the first implementation, and into \texttt{double} to get the second.

The full template definition allows for this formal nature of the \texttt{Type} typename. Using the keyword \texttt{template}, we prefix one line to our initial definition, obtaining the following function template definition:

\begin{verbatim}
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}
\end{verbatim}
In this definition we distinguish:

- **The keyword template**, starting a template definition or declaration.
- **The angle bracket enclosed list following template**. This is a list containing one or more comma-separated elements. This angle bracket enclosed list is called the *template parameter list*. Template parameter lists using multiple elements could look like this:

  ```cpp
typename Type1, typename Type2
  ```

- Inside the template parameter list we find the *formal type name* `Type`. It is a formal type name, comparable to a formal parameter name in a function's definition. Up to now we've only encountered formal variable names with functions. The *types* of the parameters were always known by the time the function was defined. Templates escalate the notion of formal names one step further up the ladder. Templates allow type names to be formalized, rather than just the variable names themselves. The fact that `Type` is a formal type name is indicated by the keyword `typename`, prefixed to `Type` in the template parameter list. A formal type name like `Type` is also called a *template type parameter*. Template non-type parameters also exist, and are shortly introduced.

Other texts on C++ sometimes use the keyword `class` where we use `typename`. So, in other texts template definitions might start with a line like:

```cpp
template <class Type>
```

In the C++ Annotations the use of `typename` over `class` is preferred, reasoning that a template type parameter is, after all, a type name (some authors prefer `class` over `typename`; in the end it's a matter of taste).

- **The template keyword and the template parameter list** is called the template header.

- The function head: it is like a normal function head, albeit that the template’s type parameters must be used in its parameter list. When the function is eventually called using actual arguments having actual types, these actual types are used by the compiler to infer which version (i.e., overload to fit the actual argument types) of the function template must be used. At the point where the function is called the compiler creates the function that is called, a process called *instantiation*. The function head may also use a formal type to specify its return value. This feature was actually used in the `add` template's definition.

- The function parameters are specified as `Type const &` parameters. This has the usual meaning: the parameters are references to `Type` objects or values that will not be modified by the function.

- The function body is like a normal function body. In the body the formal type names may be used to define or declare variables, which may then be used as any other local variable. But some restrictions apply. Looking at `add`'s body, it is clear that `operator+` is used, as well as a copy constructor, as the function returns a value. This allows us to formulate the following restrictions for the formal type `Type` as used by our `add` function template:

  - `Type` should support `operator+`
  - `Type` should support a copy constructor

  Consequently, while `Type` could be a `string`, it could never be an `ostream`, as neither `operator+` nor the copy constructor are available for streams.

Normal scope rules and identifier visibility rules apply to templates. Within the template definition’s scope formal type names overrule identically named identifiers of broader scopes.
21.1.1 Considerations regarding template parameters

We’ve managed to design our first function template:

```cpp
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}
```

Look again at `add`’s parameters. By specifying `Type const &` rather than `Type` superfluous copying is prevented, at the same time allowing values of primitive types to be passed as arguments to the function. So, when `add(3, 4)` is called, `int(4)` is assigned to `Type const &rvalue`. In general, function parameters should be defined as `Type const &` to prevent unnecessary copying. The compiler is smart enough to handle ‘references to references’ in this case, which is something the language normally does not support. For example, consider the following `main` function (here and in the following simple examples it is assumed that the template and the required headers and namespace declarations have been provided):

```cpp
int main()
{
    size_t const &var = size_t(4);
    cout << add(var, var) << '\n';
}
```

Here `var` is a reference to a constant `size_t`. It is passed as argument to `add`, thereby initializing `lvalue` and `rvalue` as `Type const &` to `size_t const &` values. The compiler interprets `Type` as `size_t`. Alternatively, the parameters might have been specified using `Type &`, rather than `Type const &`. The disadvantage of this (non-const) specification being that temporary values cannot be passed to the function anymore. The following therefore fails to compile:

```cpp
int main()
{
    cout << add(string("a"), string("b")) << '\n';
}
```

Here, a `string const &` cannot be used to initialize a `string &`. Had `add` defined `Type &` parameters then the above program would have compiled just fine. In addition the following example correctly compiles as the compiler decides that `Type` apparently is a `string const`:

```cpp
int main()
{
    string const &s = string("a");
    cout << add(s, s) << '\n';
}
```

What can we deduce from these examples?

- In general, function parameters should be specified as `Type const &` parameters to prevent unnecessary copying.
21.1. DEFINING FUNCTION TEMPLATES

- The template mechanism is fairly flexible. Formal types are interpreted as plain types, const types, pointer types, etc., depending on the actually provided types. The rule of thumb is that the formal type is used as a generic mask for the actual type, with the formal type name covering whatever part of the actual type must be covered. Some examples, assuming the parameter is defined as `Type const &`:

<table>
<thead>
<tr>
<th>Provided argument:</th>
<th>Actually used Type:</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t const</td>
<td>size_t</td>
</tr>
<tr>
<td>size_t</td>
<td>size_t</td>
</tr>
<tr>
<td>size_t *</td>
<td>size_t *</td>
</tr>
<tr>
<td>size_t const *</td>
<td>size_t const *</td>
</tr>
</tbody>
</table>

As a second example of a function template, consider the following function template:

```cpp
template <typename Type, size_t Size>
Type sum(Type const (&array)[Size])
{
    Type tp = Type();
    for (size_t idx = 0; idx < Size; idx++)
        tp += array[idx];
    return tp;
}
```

This template definition introduces the following new concepts and features:

- **The template parameter list.** This template parameter list has two elements. The first element is a well-known template type parameter, but the second element has a very specific type: a `size_t`. Template parameters of specific (i.e., non-formal) types used in template parameter lists are called **template non-type parameters**. A template non-type parameter defines the type of a constant expression, which must be known by the time the template is instantiated and which is specified in terms of existing types, such as a `size_t`.

- **Looking at the function's head, we see one parameter:**

```cpp
Type const (&array)[Size]
```

This parameter defines `array` as a reference to an array having `Size` elements of type `Type` that may not be modified.

- **In the parameter definition, both `Type` and `Size` are used.** `Type` is of course the template's type parameter `Type`, but `Size` is also a template parameter. It is a `size_t`, whose value must be inferable by the compiler when it compiles an actual call of the `sum` function template. Consequently, `Size` must be a `const` value. Such a constant expression is called a **template non-type parameter**, and its type is named in the template's parameter list.

- **When the function template is called, the compiler must be able to infer not only `Type`'s concrete value, but also `Size`'s value.** Since the function `sum` only has one parameter, the compiler is only able to infer `Size`'s value from the function's actual argument. It can do so if the provided argument is an array (of known and fixed size) rather than a pointer to `Type` elements. So, in the following `main` function the first statement will compile correctly but the second statement will not:

```cpp
int main()
```
Inside the function's body the statement `Type tp = Type()` is used to initialize `tp` to a default value. Note here that no fixed value (like 0) is used. Any type's default value may be obtained using its default constructor rather than using a fixed numeric value. Of course, not every class accepts a numeric value as an argument to one of its constructors. But all types, even the primitive types, support default constructors (actually, some classes do not implement a default constructor, or make it inaccessible; but most do). The default constructor of primitive types initializes their variables to 0 (or `false`). Furthermore, the statement `Type tp = Type()` is a true initialization: `tp` is initialized by `Type`'s default constructor, rather than using `Type`'s copy constructor to assign `Type`'s copy to `tp`.

It's interesting to note here (although not directly related to the current topic) that the syntactic construction `Type tp(Type())` cannot be used, even though it also looks like a proper initialization. Usually an initializing argument can be provided to an object's definition, like `string s("hello")`. Why, then, is `Type tp = Type()` accepted, whereas `Type tp(Type())` isn't? When `Type tp(Type())` is used it won't result in an error message. So we don't immediately detect that it's not a `Type` object's default initialization. Instead, the compiler starts generating error messages once `tp` is used. This is caused by the fact that in C++ (and in C alike) the compiler does its best to recognize a function or function pointer whenever possible: the function prevalence rule. According to this rule `Type()` is (because of the pair of parentheses) interpreted as a `pointer to a function` expecting no arguments; returning a `Type`. The compiler will do so unless it clearly isn't possible to do so. In the initialization `Type tp = Type()` it can't see a pointer to a function as a `Type` object cannot be given the value of a function pointer (remember: `Type()` is interpreted as `Type (*)()` whenever possible). But in `Type tp(Type())` it can use the pointer interpretation: `tp` is now declared as a `function` expecting a pointer to a function returning a `Type`, with `tp` itself also returning a `Type`. E.g., `tp` could have been defined as:

```cpp
Type tp(Type (*funPtr)())
{
    return (*funPtr());
}
```

Comparable to the first function template, `sum` also assumes the existence of certain public members in `Type`'s class. This time `operator+=` and `Type`'s copy constructor.

Like class definitions, template definitions should not contain `using` directives or declarations: the template might be used in a situation where such a directive overrides the programmer's intentions: ambiguities or other conflicts may result from the template's author and the programmer using different `using` directives (E.g, a `cout` variable defined in the `std` namespace and in the programmer's own namespace). Instead, within template definitions only fully qualified names, including all required namespace specifications should be used.

### 21.1.2 Late-specified return type

Traditional C++ requires function templates to specify their return type or to specify the return type as a template type parameter. Consider the following function:
21.1. DEFINING FUNCTION TEMPLATES

```cpp
int add(int lhs, int rhs)
{
    return lhs + rhs;
}
```

The above function may be converted to a function template:

```cpp
template <typename Lhs, typename Rhs>
Lhs add(Lhs lhs, Rhs rhs)
{
    return lhs + rhs;
}
```

Unfortunately, when the function template is called as

```cpp
add(3, 3.4)
```

the intended return type is probably a `double` rather than an `int`. This can be solved by adding an additional template type parameter specifying the return type but then that type must explicitly be specified:

```cpp
add<double>(3, 3.4);
```

Using `decltype` (cf. section 3.3.5) to define the return type won’t work as `lhs` and `rhs` aren’t known to the compiler by the time `decltype` is used. Thus the next attempt to get rid of the additional template type parameter fails to compile:

```cpp
template <typename Lhs, typename Rhs>
decltype(lhs + rhs) add(Lhs lhs, Rhs rhs)
{
    return lhs + rhs;
}
```

The `decltype`-based definition of a function’s return type may become fairly complex. To reduce the complexities the C++11 standard provides the `late-specified return type` syntax that *does* allow the use of `decltype` to define a function’s return type. It is primarily used with function templates but it may also be used for ordinary (non-template) functions:

```cpp
template <typename Lhs, typename Rhs>
auto add(Lhs lhs, Rhs rhs) -> decltype(lhs + rhs)
{
    return lhs + rhs;
}
```

When this function is used in a statement like `cout « add(3, 3.4)` the resulting value will be 6.4, which is most likely the intended result, rather than 6. As an example how a late-specified return type may reduce the complexity of a function’s return type definition consider the following:

```cpp
template <typename T, typename U>
decltype({*(T*)0}+{*U*}) add(T t, U u);
```
Using a late-specified return type we get the equivalent:

```c++
template <typename T, typename U>
auto add(T t, U u) -> decltype(t+u);
```

which most people think is easier to read.

The expression specified with `decltype` does not necessarily use the parameters `lhs` and `rhs` themselves. In the next function definition `lhs.length` is used instead of `lhs` itself:

```c++
template <typename Class, typename Rhs>
auto add(Class lhs, Rhs rhs) -> decltype(lhs.length() + rhs)
{
    return lhs.length() + rhs;
}
```

Any variable visible at the time `decltype` is compiled can be used in the `decltype` expression. However, it is currently not possible to handle member selection through pointers to members. The following code aims at specifying the address of a member function as `add`'s first argument and then use its return value type to determine the function template's return type:

```c++
std::string global;

template <typename MEMBER, typename RHS>
auto add(MEMBER mem, RHS rhs) -> decltype((global.*mem)() + rhs)
{
    return (global.*mem)() + rhs;
}
```

Although the above function compiles fine, it cannot currently be used as it results in a compiler error message like `unimplemented: mangling dotstar_expr` (generated for a statement like `cout << add(&string::length, 3.4)`).

### 21.2 Passing arguments by reference (reference wrappers)

Before using the reference wrappers discussed in this section the `<functional>` header file must be included.

Situations exist where the compiler is unable to infer that a reference rather than a value is passed to a function template. In the following example the function template `outer` receives `int x` as its argument and the compiler dutifully infers that `Type` is `int`:

```c++
template <typename Type>
void outer(Type t)
{
    t.x();
}
void useInt()
{
    int arg;
    outer(arg);
}
Compilation will of course fail and the compiler nicely reports the inferred type, e.g.:

```
In function 'void outer(Type) [with Type = int]': ...
```

Unfortunately the same error is generated when using `call` in the next example. The function `call` is a template expecting a function that takes an argument which is then itself modified, and a value to pass on to that function. Such a function is, e.g., `sqrtArg` expecting a reference to a `double`, which is modified by calling `std::sqrt`.

```cpp
void sqrtArg(double &arg)
{
    arg = sqrt(arg);
}
```

```cpp
template<typename Fun, typename Arg>
void call(Fun fun, Arg arg)
{
    fun(arg);
    cout << "In call: arg = " << arg << ‘\n’;
}
```

Assuming `double value = 3` then `call(sqrtArg, value)` does not modify `value` as the compiler infers `Arg` to be `double` and hence passes `value` by value.

To have `value` itself changed the compiler must be informed that `value` must be passed by reference. Note that it might not be acceptable to define `call`'s template argument as `Arg &` as not changing the actual argument might be appropriate in some situations.

The C++11 standard offers the `ref(arg)` and `cref(arg)` reference wrappers that take an argument and return it as a reference-typed argument. To actually change `value` it can be passed to `call` using `ref(value)` as shown in the following `main` function:

```cpp
int main()
{
    double value = 3;
    call(sqrtArg, value);
    cout << "Passed value, returns: " << value << ‘\n’;
    call(sqrtArg, ref(value));
    cout << "Passed ref(value), returns: " << value << ‘\n’;
    /*
     * Displays:
     * In call: arg = 1.73205
     * Passed value, returns: 3
     * In call: arg = 1.73205
     * Passed ref(value), returns: 1.73205
     */
}
```

### 21.3 Using Local and unnamed types as template arguments

Usually, types have names. But an anonymous type may also be defined:
enum
{
   V1,
   V2,
   V3
};

Here, the `enum` defines an *unnamed* or *anonymous type*.

When defining a function template, the compiler normally deduces the types of its template type parameters from its arguments:

```cpp
template <typename T>
void fun(T &&t);

fun(3); // T is int
fun('c'); // T is char
```

C++11, however, also allows the following to be used:

```cpp
fun(V1); // T is a value of the above enum type
```

Within `fun` a `T` variable may be defined, even if it's an anonymous type:

```cpp
template <typename T>
void fun(T &&t)
{
   T var(t);
}
```

C++11 also allows values or objects of locally defined types to be passed as arguments to function templates. E.g.,

```cpp
void definer()
{
   struct Local
   {
      double dVar;
      int iVar;
   };
   Local local; // using a local type
   fun(local); // OK: T is 'Local'
}
```

### 21.4 Template parameter deduction

In this section we'll concentrate on the process by which the compiler deduces the actual types of the template type parameters. These types are deduced when a function template is called using a process called *template parameter deduction*. As we've already seen, the compiler is able to substitute a wide range of actual types for a single formal template type parameter. Even so, not every
thinkable conversion is possible. In particular when a function has multiple parameters of the same template type parameter, the compiler is very restrictive when determining what argument types are actually accepted.

When the compiler deduces the actual types for template type parameters it only considers the types of the arguments that are actually used. Neither local variables nor the function's return value is considered in this process. This is understandable. When a function is called the compiler is only certain about the types of the function template's arguments. At the point of the call it definitely does not see the types of the function's local variables. Also, the function's return value might not actually be used or may be assigned to a variable of a subrange (or super-range) type of a deduced template type parameter. So, in the following example, the compiler won’t ever be able to call fun(), as it won’t be able to deduce the actual type for the Type template type parameter.

```cpp
template <typename Type>
Type fun() // can never be called as ‘fun()’
{
    return Type();
}
```

Although the compiler won’t be able to handle a call to ‘fun()’, it is possible to call fun() using an explicit type specification. E.g., fun<int>() calls fun, instantiated for int. This is of course not the same as compiler argument deduction.

In general, when a function has multiple parameters of identical template type parameters, the actual types must be exactly the same. So, whereas

```cpp
void binarg(double x, double y);
```

may be called using an int and a double, with the int argument silently being converted to a double, a similar function template cannot be called using an int and double argument: the compiler won’t by itself promote int to double deciding that Type should be double:

```cpp
template <typename Type>
void binarg(Type const &p1, Type const &p2)
{}

int main()
{
    binarg(4, 4.5); // ?? won’t compile: different actual types
}
```

What, then, are the transformations the compiler applies when deducing the actual types of template type parameters? It performs but three types of parameter type transformations and a fourth one to function template non-type parameters. If it cannot deduce the actual types using these transformations, the function template will not be considered. The transformations performed by the compiler are:

- **lvalue transformations**, creating an rvalue from an lvalue;
- **qualification transformations**, inserting a const modifier to a non-constant argument type;
- **transformation to a base class instantiated from a class template**, using a template base class when an argument of a template derived class type was provided in the call.
• Standard transformations for function template non-type parameters. This isn’t a template type parameter transformation, but it refers to any remaining template non-type parameter of function templates. For these function parameters the compiler performs any standard conversion it has available (e.g., int to size_t, int to double, etc.).

The purpose of the various template parameter type deduction transformations is not to match function arguments to function parameters, but rather, having matched arguments to parameters, to determine the actual types of the various template type parameters.

21.4.1 Lvalue transformations

There are three types of lvalue transformations:

• lvalue-to-rvalue transformations.

An lvalue-to-rvalue transformation is applied when an rvalue is required, but an lvalue is provided. This happens when a variable is used as argument to a function specifying a value parameter. For example,

```cpp
template<typename Type>
Type negate(Type value)
{
    return -value;
}

int main()
{
    int x = 5;
    x = negate(x); // lvalue (x) to rvalue (copies x)
}
```

• array-to-pointer transformations.

An array-to-pointer transformation is applied when the name of an array is assigned to a pointer variable. This is frequently used with functions defining pointer parameters. Such functions frequently receive arrays as their arguments. The array’s address is then assigned to the pointer-parameter and its type is used to deduce the corresponding template parameter’s type. For example:

```cpp
template<typename Type>
Type sum(Type *tp, size_t n)
{
    return accumulate(tp, tp + n, Type());
}

int main()
{
    int x[10];
    sum(x, 10);
}
```

In this example, the location of the array `x` is passed to `sum`, expecting a pointer to some type. Using the array-to-pointer transformation, `x`’s address is considered a pointer value which is assigned to `tp`, deducing that `Type` is `int` in the process.

• function-to-pointer transformations.

This transformation is most frequently used with function templates defining a parameter which is a pointer to a function. When calling such a function the name
of a function may be specified as its argument. The address of the function is then assigned to the pointer-parameter, deducing the template type parameter in the process. This is called a function-to-pointer transformation. For example:

```cpp
#include <cmath>

template<typename Type>
void call(Type (*fp)(Type), Type const &value)
{  
  (*fp)(value);
}

int main()
{
  call(sqrt, 2.0);
}
```

In this example, the address of the `sqrt` function is passed to `call`, expecting a pointer to a function returning a `Type` and expecting a `Type` for its argument. Using the function-to-pointer transformation, `sqrt`'s address is assigned to `fp`, deducing that `Type` is `double` in the process (note that `sqrt` is the address of a function, not a variable that is a pointer to a function, hence the lvalue transformation).

The argument `2.0` could not have been specified as `2` as there is no `int sqrt(int)` prototype. Furthermore, the function's first parameter specifies `Type (*fp)(Type)`, rather than `Type (*fp)(Type const &)` as might have been expected from our previous discussion about how to specify the types of function template's parameters, preferring references over values. However, `fp`'s argument `Type` is not a function template parameter, but a parameter of the function `fp` points to. Since `sqrt` has prototype `double sqrt(double)`, rather than `double sqrt(double const &)`, `call`'s parameter `fp` must be specified as `Type (*fp)(Type)`. It's that strict.

### 21.4.2 Qualification transformations

A qualification transformation adds `const` or `volatile` qualifications to pointers. This transformation is applied when the function template's type parameter explicitly specifies `const` (or `volatile`) but the function's argument isn't a `const` or `volatile` entity. In that case, `const` or `volatile` is provided by the compiler. Subsequently the compiler deduces the template's type parameter. For example:

```cpp
template<typename Type>
Type negate(Type const &value)
{
  return -value;
}

int main()
{
  int x = 5;
  x = negate(x);
}
```

Here we see the function template's `Type const &value` parameter: a reference to a `const Type`. However, the argument isn't a `const int`, but an `int` that can be modified. Applying a qualification transformation, the compiler adds `const` to `x`'s type, and so it matches `int const x`. This is then matched against `Type const &value` allowing the compiler to deduce that `Type` must be `int`. 
21.4.3 Transformation to a base class

Although the construction of class templates is the topic of chapter 22, we’ve already extensively used class templates before. For example, abstract containers (cf. chapter 12) are defined as class templates. Class templates can, like ordinary classes, participate in the construction of class hierarchies.

In section 22.11 it is shown how a class template can be derived from another class template. As class template derivation remains to be covered, the following discussion is necessarily somewhat premature. The reader may of course skip briefly to section 22.11 returning back to this section thereafter.

In this section it should be assumed, for the sake of argument, that a class template Vector has somehow been derived from a std::vector. Furthermore, assume that the following function template has been constructed to sort a vector using some function object obj:

```cpp
template <typename Type, typename Object>
void sortVector(std::vector<Type> vect, Object const &obj)
{
    sort(vect.begin(), vect.end(), obj);
}
```

To sort std::vector<string> objects case-insensitively, a class CaseLess could be constructed as follows:

```cpp
class CaseLess
{
    public:
        bool operator()(std::string const &before, std::string const &after) const
        {
            return strcasecmp(before.c_str(), after.c_str()) < 0;
        }
};
```

Now various vectors may be sorted using sortVector():

```cpp
int main()
{
    std::vector<string> vs;
    std::vector<int> vi;

    sortVector(vs, CaseLess());
    sortVector(vi, less<int>());
}
```

Applying the transformation transformation to a base class instantiated from a class template, the function template sortVector may now also be used to sort Vector objects. For example:

```cpp
int main()
{
    Vector<string> vs; // 'Vector' instead of 'std::vector'
```
**21.4. TEMPLATE PARAMETER DEDUCTION**

Vector<int> vi;

sortVector(vs, CaseLess());
sortVector(vi, less<int>());
}

In this example, **Vectors were passed as argument to sortVector. Applying the transformation to a base class instantiated from a class template, the compiler considers Vector to be a std::vector enabling it to deduce the template’s type parameter. A std::string for the Vector vs, an int for Vector vi.**

### 21.4.4 The template parameter deduction algorithm

The compiler uses the following algorithm to deduce the actual types of its template type parameters:

- In turn, the function template's parameters are identified using the arguments of the called function.

- For each template parameter used in the function template's parameter list, the template type parameter is associated with the corresponding argument's type (e.g., Type is int if the argument is int x, and the function's parameter is Type &value).

- While matching the argument types to the template type parameters, the three allowed transformations (see section 21.4) for template type parameters are applied where necessary.

- If identical template type parameters are used with multiple function parameters, the deduced template types must exactly match. So, the next function template cannot be called with an int and a double argument:

```cpp
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue) {
    return lvalue + rvalue;
}
```

When calling this function template, two identical types must be used (albeit that the three standard transformations are of course allowed). If the template deduction mechanism does not come up with identical actual types for identical template types, then the function template is not going to be instantiated.

### 21.4.5 Template type contractions

With function templates the combination of the types of template arguments and template parameters shows some interesting contractions. What happens, for example if a template type parameter is specified as an rvalue reference but an lvalue reference argument type is provided?

In such cases the compiler performs type contractions. Doubling identical reference types results in a simple contraction: the type is deduced to be a single reference type. Example: if the template parameter type is specified as a Type && and the actual parameter is an int && then Type is deduced to be an int, rather than an int & &.
This is fairly intuitive. But what happens if the actual type is `int &`? There is no such thing as an `int & &&param` and so the compiler contracts the double reference by removing the rvalue reference, keeping the lvalue reference. Here the following rules are applied:

1. A function template parameter defined as an lvalue reference to a template’s type parameter (e.g., `Type &`) receiving an lvalue reference argument results in a single lvalue reference.

2. A function template parameter defined as an rvalue reference to a template’s type parameter (e.g., `Type &&`) receiving any kind of reference argument uses the reference type of the argument.

Examples:

- When providing an `Actual &` argument then `Type &` becomes an `Actual &` and `Type` is inferred as `Actual`;
- When providing an `Actual &` then `Type &&` becomes an `Actual &` and `Type` is inferred as `Actual`;
- When providing an `Actual &` then `Type` also becomes `Actual &` and `Type` is inferred as `Actual`;
- When providing an `Actual &&` then `Type &&` becomes `Actual &&` and `Type` is inferred as `Actual`;

Let’s look at a concrete example where contraction occurs. Consider the following function template where a function parameter is defined as an rvalue reference to some template type parameter:

```cpp
template <typename Type>
void function(Type &&param)
{
    callee(static_cast<Type &&>(param));
}
```

In this situation, when `function` is called with an (lvalue) argument of type `TP &` the template type parameter `Type` is deduced to be `Tp &`. Therefore, `Type &&param` is instantiated as `Tp &&param`, `Type` becomes `Tp` and the rvalue reference is replaced by an lvalue reference.

Likewise, when `callee` is called using the `static_cast` the same contraction occurs, so `Type &&param` operates on `Tp &&param`. Therefore (using contraction) the static cast also uses type `Tp &&param`. If `param` happened to be of type `Tp &&` then the static cast uses type `Tp &&param`.

This characteristic allows us to pass a function argument to a nested function without changing its type: lvalues remain lvalues, rvalues remain rvalues. This characteristic is therefore also known as perfect forwarding which is discussed in greater detail in section 22.5.2. Perfect forwarding prevents the template author from having to define multiply overloaded versions of a function template.

### 21.5 Declaring function templates

Up to now, we’ve only defined function templates. There are various consequences of including function template definitions in multiple source files, none of them serious, but worth knowing.
• Like class interfaces, template definitions are usually included in header files. Every time a header file containing a template definition is read by the compiler it must process the full definition. It must do so even if it does not actually use the template. This somewhat slows-down the compilation. For example, compiling a template header file like `algorithm` on my old laptop takes about four times the amount of time it takes to compile a plain header file like `cmath`. The header file `iostream` is even harder to process, requiring almost 15 times the amount of time it takes to process `cmath`. Clearly, processing templates is serious business for the compiler. On the other hand this drawback shouldn’t be taken too seriously. Compilers are continuously improving their template processing capacity and computers keep getting faster and faster. What was a nuisance a few years ago is hardly noticeable today.

• Every time a function template is instantiated, its code appears in the resulting object module. However, if multiple instantiations of a template using the same actual types for its template parameters exist in multiple object files the one definition rule is lifted. The linker weeds out superfluous instantiations (i.e., identical definitions of instantiated templates). In the final program only one instantiation for a particular set of actual template type parameters remain available (see section 21.6 for an illustration). Therefore, the linker has an additional task to perform (viz. weeding out multiple instantiations), which somewhat slows down the linking process.

• Sometimes the definitions themselves are not required, but references or pointers to the templates are. Requiring the compiler to process the full template definitions in those cases needlessly slows down the compilation process.

• In the context of template meta programming (see chapter 23) it is sometimes not even required to provide a template implementation. Instead, only specializations (cf. section 21.9) are created which are based upon the mere declaration.

So in some contexts template definitions may not be required. Instead the software engineer may opt to declare a template rather than to include the template’s definition time and again in various source files.

When templates are declared, the compiler does not have to process the template’s definitions again and again; and no instantiations are created on the basis of template declarations alone. Any actually required instantiation must then be available elsewhere (of course, this holds true for declarations in general). Unlike the situation we encounter with ordinary functions, which are usually stored in libraries, it is currently not possible to store templates in libraries (although the compiler may construct precompiled header files). Consequently, using template declarations puts a burden on the shoulders of the software engineer, who has to make sure that the required instantiations exist. Below a simple way to accomplish that is introduced.

To create a function template declaration simply replace the function’s body by a semicolon. Note that this is exactly identical to the way ordinary function declarations are constructed. So, the previously defined function template `add` can simply be declared as

```cpp
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue);
```

We’ve already encountered template declarations. The header file `iosfwd` may be included in sources not requiring instantiations of elements from the class `ios` and its derived classes. For example, to compile the declaration

```cpp
std::string getCsvLine(std::istream &in, char const *delim);
```

it is not necessary to include the `string` and `istream` header files. Rather, a single
#include <iosfwd>

is sufficient. Processing iosfwd requires only a fraction of the time it takes to process the string and istream header files.

## 21.5.1 Instantiation declarations

If declaring function templates speeds up the compilation and the linking phases of a program, how can we make sure that the required instantiations of the function templates are available when the program is eventually linked together?

For this a variant of a template declaration is available, a so-called **explicit instantiation declaration**. An explicit instantiation declaration consists of the following elements:

- It starts with the keyword `template`, omitting the template parameter list.
- Next the function template’s return type and name are specified.
- The function template’s name is followed by a **type specification list**. A type specification list is an angle brackets enclosed list of type names. Each type specifies the actual type of the corresponding template type parameter in the template’s parameter list.
- Finally the function template’s parameter list is specified, terminated by a semicolon.

Although this is a declaration, it is understood by the compiler as a request to instantiate that particular variant of the function template.

Using explicit instantiation declarations all instantiations of template functions required by a program can be collected in one file. This file, which should be a normal source file, should include the template definition header file and should subsequently specify the required explicit instantiation declarations. Since it's a source file, it is not included by other sources. So namespace using directives and declarations may safely be used once the required headers have been included. Here is an example showing the required instantiations for our earlier `add` function template, instantiated for `double`, `int`, and `std::string` types:

```cpp
#include "add.h"
#include <string>
using namespace std;

template int add<int>(int const &lvalue, int const &rvalue);
template double add<double>(double const &lvalue, double const &rvalue);
template string add<string>(string const &lvalue, string const &rvalue);
```

If we’re sloppy and forget to mention an instantiation required by our program then the repair is easily made by adding the missing instantiation declaration to the above list. After recompiling the file and relinking the program we’re done.

## 21.6 Instantiating function templates

Different from an ordinary function that results in code once the compiler reads its definition a template is not instantiated when its definition is read. A template is merely a **recipe** telling the
compiler how to create particular code once it’s time to do so. It’s indeed very much like a recipe in a cooking book. You reading how to bake a cake doesn’t mean you have actually baked that cake by the time you’ve read the recipe.

So, when is a function template actually instantiated? There are two situations where the compiler decides to instantiate templates:

- They are instantiated when they are used (e.g., the function `add` is called with a pair of `size_t` values);
- When addresses of function templates are taken they are instantiated. Example:

```cpp
char (*addptr)(char const &, char const &) = add;
```

The location of statements causing the compiler to instantiate a template is called the template’s point of instantiation. The point of instantiation has serious implications for the function template’s code. These implications are discussed in section 21.13.

The compiler is not always able to deduce the template’s type parameters unambiguously. When the compiler reports an ambiguity it must be solved by the software engineer. Consider the following code:

```cpp
#include <iostream>
#include "add.h"

size_t fun(int (*)(int*, size_t n));
double fun(double (*)(double*, size_t n));

int main()
{
    std::cout << fun(add);
}
```

When this little program is compiled, the compiler reports an ambiguity it cannot resolve. It has two candidate functions as for each overloaded version of `fun` an add function can be instantiated:

```cpp
error: call of overloaded 'fun(<unknown type>)' is ambiguous
note: candidates are: int fun(size_t (*)(int*, size_t))
note: double fun(double (*)(double*, size_t))
```

Such situations should of course be avoided. Function templates can only be instantiated if there’s no ambiguity. Ambiguities arise when multiple functions emerge from the compiler’s function selection mechanism (see section 21.14). It is up to us to resolve the ambiguities. They could be resolved using a blunt `static_cast` (by which we select among alternatives, all of them possible and available):

```cpp
#include <iostream>
#include "add.h"

int fun(int (*)(int const &lvalue, int const &rvalue));
double fun(double (*)(double const &lvalue, double const &rvalue));

int main()
{
    
```
std::cout << fun(
    static_cast<int (*)(int const &, int const &)> (add)
);
21.7 Using explicit template types

In the previous section we saw that the compiler may encounter ambiguities when attempting to instantiate a template. In an example overloaded versions of a function (`fun`) existed, expecting different types of arguments. The ambiguity resulted from the fact that both arguments could have been provided by an instantiation of a function template. The intuitive way to solve such an ambiguity is to use a `static_cast`. But casts should be avoided wherever possible.

With function templates static casts may indeed be avoided using `explicit template type arguments`. Explicit template type arguments can be used to inform the compiler about the actual types it should use when instantiating a template. To use explicit type arguments the function’s name is followed by an `actual template type argument list` which may again be followed by the function’s argument list. The actual types mentioned in the actual template argument list are used by the compiler to ‘deduce’ what types to use when instantiating the template. Here is the example from the previous section, now using explicit template type arguments:

```c++
#include "add.h"
#include "pointerunion.h"

void fun();

int main()
{
    PointerUnion pu = { add };
    fun();
    std::cout << pu.vp << 'n';
}

• All sources are compiled to object modules. Note the different sizes of `source1.o` (1912 bytes using `g++` version 4.3.4 (sizes of object modules reported in this section may differ for different compilers and/or run-time libraries)) and `source2.o` (1740 bytes). Since `source1.o` contains the instantiation of `add`, it is somewhat larger than `source2.o`, containing only the template’s declaration. Now we’re ready to start our little experiment.

• Linking `main.o` and `source1.o`, we obviously link together two object modules, each containing its own instantiation of the same template function. The resulting program produces the following output:

```
0x80486d8
0x80486d8
```

Furthermore, the size of the resulting program is 6352 bytes.

• Linking `main.o` and `source2.o`, we now link together an object module containing the instantiation of the `add` template, and another object module containing the mere declaration of the same template function. So, the resulting program cannot but contain a single instantiation of the required function template. This program has exactly the same size, and produces exactly the same output as the first program.

From our little experiment we conclude that the linker indeed removes identical template instantiations from a final program. Furthermore we conclude that using mere template declarations does not result in template instantiations.

21.7 Using explicit template types
Explicit template type arguments can be used in situations where the compiler has no way to detect which types should actually be used. E.g., in section 21.4 the function template Type fun() was defined. To instantiate this function for the double type, we can call fun<double>().

21.8 Overloading function templates

Let’s once again look at our add template. That template was designed to return the sum of two entities. If we would want to compute the sum of three entities, we could write:

```cpp
int main()
{
    add(add(2, 3), 4);
}
```

This is an acceptable solution for the occasional situation. However, if we would have to add three entities regularly, an overloaded version of the add function expecting three arguments might be a useful function to have. There’s a simple solution to this problem: function templates may be overloaded.

To define an overloaded function template, merely put multiple definitions of the template in its header file. For the add function this would boil down to:

```cpp
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}
template <typename Type>
Type add(Type const &lvalue, Type const &mvalue, Type const &rvalue)
{
    return lvalue + mvalue + rvalue;
}
```

The overloaded function does not have to be defined in terms of simple values. Like all overloaded functions, a unique set of function parameters is enough to define an overloaded function template. For example, here’s an overloaded version that can be used to compute the sum of the elements of a vector:

```cpp
template <typename Type>
Type add(std::vector<Type> const &vect)
When overloading function templates we do not have to restrict ourselves to the function’s parameter list. The template’s type parameter list itself may also be overloaded. The last definition of the add template allows us to specify a vector as its first argument, but no deque or map. Overloaded versions for those types of containers could of course be constructed, but how far should we go? A better approach seems to be to look for common characteristics of these containers. If found we may be able to define an overloaded function template based on these common characteristics. One common characteristic of the mentioned containers is that they all support begin and end members, returning iterators. Using this, we could define a template type parameter representing containers that must support these members. But mentioning a plain ‘container type’ doesn’t tell us for what type of data it was instantiated. So we need a second template type parameter representing the container’s data type, thus overloading the template’s type parameter list. Here is the resulting overloaded version of the add template:

```cpp
template <typename Container, typename Type>
Type add(Container const &cont, Type const &init)
{
    return std::accumulate(cont.begin(), cont.end(), init);
}
```

One may wonder whether the init parameter could not be left out of the parameter list as init often has a default initialization value. The answer is ‘yes’, but there are complications. It is possible to define the add function as follows:

```cpp
template <typename Type, typename Container>
Type add(Container const &cont)
{
    return std::accumulate(cont.begin(), cont.end(), Type());
}
```

Note, however, that the template’s type parameters were reordered, which is necessary because the compiler won’t be able to determine Type in a call like:

```cpp
int x = add(vectorOfInts);
```

After reordering the template type parameters, putting Type first, an explicit template type argument can be provided for the first template type parameter:

```cpp
int x = add<int>(vectorOfInts);
```

In this example we provided a vector<int> argument. One might wonder why we have to specify int explicitly to allow the compiler to determine the template type parameter Type. In fact, we don’t. A third kind of template parameter exists, a template template parameter, allowing the compiler to determine Type directly from the actual container argument. Template template parameters are discussed in section 23.4.
21.8.1 An example using overloaded function templates

With all these overloaded versions in place, we may now start the compiler to compile the following function:

```cpp
using namespace std;

int main()
{
    vector<int> v;

    add(3, 4); // 1 (see text)
    add(v); // 2
    add(v, 0); // 3
}
```

- In statement 1 the compiler recognizes two identical types, both `int`. It therefore instantiates `add<int>`, our very first definition of the `add` template.
- In statement 2 a single argument is used. Consequently, the compiler looks for an overloaded version of `add` requiring but one argument. It finds the overloaded function template expecting a `std::vector`, deducing that the template’s type parameter must be `int`. It instantiates `add<int>(std::vector<int> const &)`.
- In statement 3 the compiler again encounters an argument list having two arguments. However, this time the types of the arguments aren’t equal, so `add` template’s first definition can’t be used. But it can use the last definition, expecting entities having different types. As a `std::vector` supports `begin` and `end`, the compiler is now able to instantiate the function template `add<std::vector<int>, int>(std::vector<int> const &, int const &)`.

Having defined the `add` function template for two equal and two different template type parameters we’ve exhausted the possibilities for using an `add` function template having two template type parameters.

21.8.2 Ambiguities when overloading function templates

Although it is possible to define another function template `add` this introduces an ambiguity as the compiler won’t be able to choose which of the two overloaded versions defining two differently typed function parameters should be used. For example when defining:

```cpp
#include "add.h"

template <typename T1, typename T2>
T1 add(T1 const &lvalue, T2 const &rvalue)
{
    return lvalue + rvalue;
}
int main()
{
```
21.8. OVERLOADING FUNCTION TEMPLATES

```cpp
add(3, 4.5);
```

the compiler reports an ambiguity like the following:

```
error: call of overloaded ‘add(int, double)’ is ambiguous
error: candidates are: Type add(const Container&, const Type&)
    [with Container = int, Type = double]
error: Type1 add(const T1&, const T2&)
    [with T1 = int, T2 = double]
```

Now recall the overloaded function template accepting three arguments:

```cpp
template <typename Type>
Type add(Type const &lvalue, Type const &mvalue, Type const &rvalue)
{
    return lvalue + mvalue + rvalue;
}
```

It may be considered as a disadvantage that only equally typed arguments are accepted by this function (three `int`s, three `double`s, etc.). To remedy this we define yet another overloaded function template, this time accepting arguments of any type. This function template can only be used if `operator+` is defined between the function's actually used types, but apart from that there appears to be no problem. Here is the overloaded version accepting arguments of any type:

```cpp
template <typename Type1, typename Type2, typename Type3>
Type1 add(Type1 const &lvalue, Type2 const &mvalue, Type3 const &rvalue)
{
    return lvalue + mvalue + rvalue;
}
```

Now that we've defined the above two overloaded function templates expecting three arguments let's call `add` as follows:

```cpp
add(1, 2, 3);
```

Should we expect an ambiguity here? After all, the compiler might select the former function, deducing that `Type == int`, but it might also select the latter function, deducing that `Type1 == int`, `Type2 == int` and `Type3 == int`. Remarkably, the compiler reports no ambiguity.

No ambiguity is reported because of the following. If overloaded template functions are defined using less and more specialized template type parameters (e.g., less specialized: all types different vs. more specialized: all types equal) then the compiler selects the more specialized function whenever possible.

As a rule of thumb: overloaded function templates must allow a unique combination of template type arguments to be specified to prevent ambiguities when selecting which overloaded function template to instantiate. The ordering of template type parameters in the function template's type parameter list is not important. E.g., trying to instantiate one of the following function templates results in an ambiguity:

```cpp
template <typename T1, typename T2>
```
21.8.3 Declaring overloaded function templates

Like any function, overloaded functions may be declared, either using plain declarations or instantiation declarations. Explicit template argument types may also be used. Example:

- To declare a function template `add` accepting certain containers:

```cpp
template <typename Container, typename Type>
Type add(Container const &container, Type const &init);
```

- to use an instantiation declaration (in which case the compiler must already have seen the template's definition):

```cpp
template int add<std::vector<int>, int>
(std::vector<int> const &vect, int const &init);
```

- to use explicit template type arguments:

```cpp
std::vector<int> vi;
int sum = add<std::vector<int>, int>(vi, 0);
```

21.9 Specializing templates for deviating types

The initial `add` template, defining two identically typed parameters works fine for all types supporting `operator+` and a copy constructor. However, these assumptions are not always met. For example, with `char *s`, using `operator+` or a ’copy constructor’ does not make sense. The compiler tries to instantiate the function template, but compilation fails as `operator+` is not defined for pointers.

In such situations the compiler may be able to resolve the template type parameters but it (or we ...) may then detect that the standard implementation is pointless or produces errors.

To solve this problem a template explicit specialization may be defined. A template explicit specialization defines the function template for which a generic definition already exists using specific actual template type parameters. As we saw in the previous section the compiler always prefers a more specialized function over a less specialized one. So the template explicit specialization is selected whenever possible.

A template explicit specialization offers a specialization for its template type parameter(s). The special type is consistently substituted for the template type parameter in the function template’s code. For example if the explicitly specialized type is `char const *` then in the template definition

```cpp
template <typename Type>
```
Type add(Type const &lvalue, Type const &rvalue) {
    return lvalue + rvalue;
}

Type must be replaced by char const *, resulting in a function having prototype

char const *add(char const *const &lvalue, char const *const &rvalue);

Now we try to use this function:

int main(int argc, char **argv) {
    add(argv[0], argv[1]);
}

However, the compiler ignores our specialization and tries to instantiate the initial function template. This fails, leaving us wondering why it didn’t select the explicit specialization....

To see what happened here we replay, step by step, the compiler’s actions:

- add is called with char * arguments.
- Both types are equal, so the compiler deduces that Type equals char *.
- Now it inspects the specialization. Can a char * template type argument match a char const *const & template parameter? Here opportunities for the allowable transformations from section 21.4 may arise. A qualification transformation seems to be the only viable one, allowing the compiler to bind a const-parameter to a non-const argument.
- So, in terms of Type the compiler can match an argument of some Type or an argument of some Type const to a Type const &.
- Type itself is not modified, and so Type is a char *.
- Next the compiler inspects the available explicit specializations. It finds one, specializing for char const *.
- Since a char const * is not a char * it rejects the explicit specialization and uses the generic form, resulting in a compilation error.

If our add function template should also be able to handle char * template type arguments another explicit specialization for char * may be required, resulting in the prototype

char *add(char *const &lvalue, char *const &rvalue);

Instead of defining another explicit specialization an overloaded function template could be designed expecting pointers. The following function template definition expects two pointers to constant Type values and returns a pointer to a non-constant Type:

template <typename Type>
Type *add(Type const *t1, Type const *t2) {
    std::cout << "Pointers\n";
    return new Type;
}
What actual types may be bound to the above function parameters? In this case only a `Type const *`, allowing `char const *`’s to be passed as arguments. There’s no opportunity for a qualification transformation here. The qualification transformation allows the compiler to add a `const` to a non-const argument if the parameter itself (and not `Type`) is specified in terms of a `const` or `const &`. Looking at, e.g., `t1` we see that it’s defined as a `Type const *`. There’s nothing `const` here that’s referring to the parameter (in which case it would have been `Type const *const t1` or `Type const *const &t1`). Consequently a qualification transformation cannot be applied here.

As the above overloaded function template only accepts `char const *` arguments, it will not accept (without a reinterpret cast) `char *` arguments. So `main`’s `argv` elements cannot be passed to our overloaded function template.

### 21.9.1 Avoiding too many specializations

So do we have to define yet another overloaded function template, this time expecting `Type *` arguments? It is possible, but at some point it should become clear that our approach doesn’t scale. Like ordinary functions and classes, function templates should have one conceptually clear purpose. Trying to add overloaded function templates to overloaded function templates quickly turns the template into a kludge. Don’t use this approach. A better approach is to construct the template so that it fits its original purpose, to make allowances for the occasional specific case and to describe its purpose clearly in its documentation.

In some situations constructing template explicit specialization may of course be defensible. Two specializations for `const` and non-`const` pointers to characters might be appropriate for our `add` function template. Here’s how they are constructed:

- Start with the keyword `template`.
- Next, an empty set of angle brackets is written. This indicates to the compiler that there must be an `existing` template whose prototype matches the one we’re about to define. If we err and there is no such template then the compiler reports an error like:

  ```
  error: template-id ‘add<char*>' for ‘char* add(char* const&, char* const&)' does not match any template declaration
  ```

- Now the function’s head is defined. It must match the prototype of the initial function template or the form of a template explicit instantiation declaration (see section 21.5.1) if its specialized type cannot be determined from the function’s arguments. It must specify the correct return type, function name, maybe explicit template type arguments, as well as the function’s parameter list.
- Finally the function’s body is defined, providing the special implementation that is required for the specialization.

Here are two explicit specializations for the function template `add`, expecting `char *` and `char const *` arguments:

```cpp
template <> char *add<char *>(char *const &p1,
                           char *const &p2)
{
  std::string str(p1);
  str += p2;
  return strcpy(new char[str.length() + 1], str.c_str());
}
```


```cpp
template <> char const* add(char const *const &p1, char const *const &p2)
{
    static std::string str;
    str = p1;
    str += p2;
    return str.c_str();
}
```

Template explicit specializations are normally included in the file containing the other function template's implementations.

### 21.9.2 Declaring specializations

Template explicit specializations can be declared in the usual way. I.e., by replacing its body with a semicolon.

When declaring a template explicit specialization the pair of angle brackets following the `template` keyword are essential. If omitted, we would have constructed a template instantiation declaration. The compiler would silently process it, at the expense of a somewhat longer compilation time.

When declaring a template explicit specialization (or when using an instantiation declaration) the explicit specification of the template type parameters can be omitted if the compiler is able to deduce these types from the function's arguments. As this is the case with the `char (const) *` specializations, they could also be declared as follows:

```cpp
template <> char * add(char *const &p1, char *const &p2)
template <> char const * add(char const *const &p1, char const *const &p2);
```

If in addition `template <>` could be omitted the template character would be removed from the declaration. The resulting declaration is now a mere function declaration. This is not an error: function templates and ordinary (non-template) functions may mutually overload each other. Ordinary functions are not as restrictive as function templates with respect to allowed type conversions. This could be a reason to overload a template with an ordinary function every once in a while.

A function template explicit specialization is not just another overloaded version of the function template. Whereas an overloaded version may define a completely different set of template parameters, a specialization must use the same set of template parameters as its non-specialized variant. The compiler uses the specialization in situations where the actual template arguments match the types defined by the specialization (following the rule that the most specialized set of parameters matching a set of arguments will be used). For different sets of parameters overloaded versions of functions (or function templates) must be used.

### 21.9.3 Complications when using the insertion operator

Now that we've covered explicit specializations and overloading let's consider what happens when a class defines a `std::string` conversion operator (cf. section 11.3).

A conversion operator is guaranteed to be used as an rvalue. This means that objects of a class defining a `string` conversion operator can be assigned to, e.g., `string` objects. But when trying to
insert objects defining string conversion operators into streams then the compiler complains that we're attempting to insert an inappropriate type into an ostream.

On the other hand, when this class defines an int conversion operator insertion is performed flawlessly.

The reason for this distinction is that operator« is defined as a plain (free) function when inserting a basic type (like int) but it is defined as a function template when inserting a string. Hence, when trying to insert an object of our class defining a string conversion operator the compiler visits all overloaded versions of insertion operators inserting into ostream objects.

Since no basic type conversion is available the basic type insertion operators can't be used. Since the available conversions for template arguments do not allow the compiler to look for conversion operators our class defining the string conversion operator cannot be inserted into an ostream.

If it should be possible to insert objects of such a class into ostream objects the class must define its own overloaded insertion operator (in addition to the string conversion operator that was required to use the class's objects as rvalue in string assignments).

21.10 Static assertions

The

```cpp
static_assert(constant expression, error message)
```

utility is defined by the C++11 standard to allow assertions to be made within template definitions. Here are two examples of its use:

```cpp
static_assert(BUFSIZE1 == BUFSIZE2,
              "BUFSIZE1 and BUFSIZE2 must be equal");

template <typename Type1, typename Type2>
void rawswap(Type1 &type1, Type2 &type2)
{
    static_assert(sizeof(Type1) == sizeof(Type2),
                  "rawswap: Type1 and Type2 must have equal sizes");
    // ...
}
```

The first example shows how to avoid yet another preprocessor directive (in this case the #error directive).

The second example shows how static_assert can be used to ensure that a template operates under the right condition(s).

The string defined in static_assert's second argument is displayed and compilation stops if the condition specified in static_assert's first argument is false.

Like the #error preprocessor directive static_assert is a compile-time matter that doesn't have any effect on the run-time efficiency of the code in which it is used.
21.11 Numeric limits

The header file `<climits>` defines constants for various types, e.g., `INT_MAX` defines the maximum value that can be stored in an `int`.

The disadvantage of the limits defined in `<climits>` is that they are fixed limits. Let’s assume you write a function template that receives an argument of a certain type. E.g.,

```cpp
template<typename Type>
Type operation(Type &&type);
```

Assume this function should return the largest negative value for `Type` if `type` is a negative value and the largest positive value if `type` is a positive value. However, 0 should be returned if the type is not an integral value.

How to proceed?

Since the constants in `<climits>` can only be used if the type to use is already known, the only approach seems to be to create function template specializations for the various integral types, like:

```cpp
template<>
int operation<int>(int &&type)
{
    return type < 0 ? INT_MIN : INT_MAX;
}
```

The facilities provided by numeric limits provide an alternative. To use these facilities the header file `<limits>` header file must be included.

The class template `numeric_limits` offers various members answering all kinds of questions that could be asked of numeric types. Before introducing these members, let’s have a look at how we could implement the `operation` function template as just one single function template:

```cpp
template<typename Type>
Type operation(Type &&type)
{
    return
    not numeric_limits<Type>::is_integer ? 0 :
    type < 0 ? numeric_limits<Type>::min() :
        numeric_limits<Type>::max();
}
```

Now `operation` can be used for all the language’s primitive types.

Here is an overview of the facilities offered by `numeric_limits`. Note that the member functions defined by `numeric_limits` return `constexpr` values. A member ‘member’ defined by `numeric_limits` for type `Type` can be used as follows:

```cpp
numeric_limits<Type>::member // data members
numeric_limits<Type>::member() // member functions
```

All `numeric_limits` member functions return `constexpr` values.
### CHAPTER 21. FUNCTION TEMPLATES

- **Type** `denorm_min()`:
  
  *Available for Type:* its minimum positive denormalized value; otherwise it returns `numeric_limits<Type>::min()`.

- **int** `digits`:

  The number of non-sign bits used by `Type` values, or (floating point types) the number of digits in the mantissa are returned.

- **int** `digits10`:

  The number of digits that are required to represent a `Type` value without changing it.

- **Type** `constexpr epsilon()`:

  The difference for `Type` between the smallest value exceeding 1 and 1 itself.

- **float_denorm_style** `has_denorm`:

  Denormalized floating point value representations use a variable number of exponent bits. The `has_denorm` member returns information about denormalized values for type `Type`:
  
  - `denorm_absent`: `Type` does not allow denormalized values;
  
  - `denorm_indeterminate`: `Type` may or may not use denormalized values; the compiler cannot determine this at compile-time;
  
  - `denorm_present`: `Type` uses denormalized values;

- **bool** `has_denorm_loss`:

  True if a loss of accuracy was detected as a result of using denormalization (rather than being an inexact result).

- **bool** `has_infinity`:

  True if `Type` has a representation for positive infinity.

- **bool** `has_quiet_NaN`:

  True if `Type` has a representation for a non-signaling ‘Not-a-Number’ value.

- **bool** `has_signaling_NaN`:

  True if `Type` has a representation for a signaling ‘Not-a-Number’ value.

- **Type** `constexpr infinity()`:

  If available for `Type`: its positive infinity value.

- **bool** `is_bounded`:

  True if `Type` contains a finite set of values.

- **bool** `is_exact`:

  True if `Type` uses an exact representation.

- **bool** `is_iec559`:

  True if `Type` uses the IEC-559 (IEEE-754) standard. Such types always return true for `has_infinity`, `has_quiet_NaN` and `has_signaling_NaN`, while `infinity()`, `quiet_NaN()` and `signaling_NaN()` return non-zero values.
21.11. NUMERIC LIMITS

- **bool is_integer**: 
  true if `Type` is an integral type.

- **bool is_modulo**: 
  true if `Type` is a ‘modulo’ type. Values of modulo types can always be added, but the addition may ‘wrap around’ producing a smaller `Type` result than either of the addition’s two operands.

- **bool is_signed**: 
  true if `Type` is signed.

- **bool is_specialized**: 
  true for specializations of `Type`.

- **Type constexpr lowest()**: 
  `Type`’s lowest finite value representable by `Type`: no other finite value smaller than the value returned by `lowest` exists. This value equals the value returned by `min` except for floating-point types.

- **T constexpr max()**: 
  `Type`’s maximum value.

- **T constexpr min()**: 
  `Type`’s minimum value. For denormalized floating point types the minimum positive normalized value.

- **int max_exponent**: 
  maximum positive integral value for the exponent of a floating point type `Type` producing a valid `Type` value.

- **int max_exponent10**: 
  maximum integral value for the exponent that can be used for base 10, producing a valid `Type` value.

- **int min_exponent**: 
  minimum negative integral value for the exponent of a floating point type `Type` producing a valid `Type` value.

- **int min_exponent10**: 
  minimum negative integral value for the exponent that can be used for base 10, producing a valid `Type` value.

- **Type constexpr quiet_NaN()**: 
  if available for `Type`: its a non-signaling ‘Not-a-Number’ value.

- **int radix**: 
  if `Type` is an integral type: base of the representation; if `Type` is a floating point type: the base of the exponent of the representation.
• Type constexpr round_error():
  the maximum rounding error for Type.

• float round_style round_style:
  the rounding style used by Type. It has one of the following enum float round_style values:
  – round_toward_zero: values are rounded toward zero;
  – round_to_nearest: values are rounded to the nearest representable value;
  – round_toward_infinity: values are rounded toward infinity;
  – round_toward_neg_infinity: if it rounds toward negative infinity;
  – round_indeterminate: if the rounding style is indeterminable at compile-time.

• Type constexpr signaling_NaN():
  if available for Type: its a signaling 'Not-a-Number' value.

• bool tinyness_before:
  true if Type allows tinyness to be detected before rounding.

• bool traps:
  true if Type implements trapping.

### 21.12 Polymorphous wrappers for function objects

In C++ pointers to (member) functions have fairly strict rvalues. They can only point to functions matching their types. This becomes a problem when defining templates where the type of a function pointer may depend on the template’s parameters.

To solve this problem the C++11 standard introduces polymorphous (function object) wrappers. Polymorphous wrappers refer to function pointers, member functions or function objects, as long as their parameters match in type and number.

Before using polymorphic function wrappers the header file `<functional>` must be included.

Polymorphic function wrappers are made available through the `std::function` class template. Its template argument is the prototype of the function to create a wrapper for. Here is an example of the definition of a polymorphic function wrapper that can be used to point to a function expecting two int values and returning an int:

```cpp
std::function<int (int, int)> ptr2fun;
```

Here, the template’s parameter is `int (int, int)`, indicating a function expecting two int arguments, and returning an int. Other prototypes return other, matching, function wrappers.

Such a function wrapper can now be used to point to any function the wrapper was created for. E.g., ‘`plus<int>` add’ creates a functor defining an `int operator()(int, int)` function call member. As this qualifies as a function having prototype `int (int, int)`, our `ptr2fun` may point to add:

```cpp
ptr2fun = add;
```
If `ptr2fun` does not yet point to a function (e.g., it is merely defined) and an attempt is made to call a function through it a `std::bad_function_call` exception is thrown. Also, a polymorphic function wrapper that hasn’t been assigned to a function’s address represents the value `false` in logical expressions (as if it had been a pointer having value zero):

```cpp
std::function<int(int)> ptr2int;
if (not ptr2int)
    cout << "ptr2int is not yet pointing to a function\n";
```

Polymorphous function wrappers can also be used to refer to functions, functors or other polymorphous function wrappers having prototypes for which standard conversions exist for either parameters or return values. E.g.,

```cpp
bool predicate(long long value);
void demo()
{
    std::function<int(int)> ptr2int;
    ptr2int = predicate;  // OK, convertible param. and return type
    struct Local
    {
        short operator()(char ch);
    };
    Local object;
    std::function<short(char)> ptr2char(object);
    ptr2int = object;  // OK, object is a functor whose function
                      // operator has a convertible param. and
                      // return type.
    ptr2int = ptr2char;  // OK, now using a polym. funct. wrapper
}
```

### 21.13 Compiling template definitions and instantiations

Consider this definition of the `add` function template:

```cpp
template <typename Container, typename Type>
Type add(Container const &container, Type init)
{
    return std::accumulate(container.begin(), container.end(), init);
}
```

Here `std::accumulate` is called using `container`'s `begin` and `end` members.

The calls `container.begin()` and `container.end()` are said to depend on template type parameters. The compiler, not having seen `container`'s interface, cannot check whether `container` actually has members `begin` and `end` returning input iterators.
On the other hand, \texttt{std::accumulate} itself is independent of any template type parameter. Its \textit{arguments} depend on template parameters, but the function call itself isn’t. Statements in a template’s body that are independent of template type parameters are said \textit{not to depend on template type parameters}.

When the compiler encounters a template definition, it verifies the syntactic correctness of all statements not depending on template parameters. I.e., it must have seen all class definitions, all type definitions, all function declarations etc. that are used in those statements. If the compiler hasn’t seen the required definitions and declarations then it will reject the template’s definition. Therefore, when submitting the above template to the compiler the header file \texttt{numeric} must first have been included as this header file declares \texttt{std::accumulate}.

With statements depending on template parameters the compiler cannot perform those extensive syntactic checks. It has no way to verify the existence of a member \texttt{begin} for the as yet unspecified type \texttt{Container}. In these cases the compiler performs superficial checks, assuming that the required members, operators and types eventually become available.

The location in the program’s source where the template is instantiated is called its \textit{point of instantiation}. At the point of instantiation the compiler deduces the actual types of the template’s parameters. At that point it checks the syntactic correctness of the template’s statements that depend on template type parameters. This implies that the compiler must have seen the required declarations \textit{only at the point of instantiation}. As a rule of thumb, you should make sure that all required declarations (usually: header files) have been read by the compiler at every point of instantiation of the template. For the template’s definition itself a more relaxed requirement can be formulated. When the definition is read only the declarations required for statements \textit{not} depending on the template’s type parameters must have been provided.

### 21.14 The function selection mechanism

When the compiler encounters a function call, it must decide which function to call when overloaded functions are available. Earlier we’ve encountered principles like ‘the most specific function is selected’. This is a fairly intuitive description of the compiler’s function selection mechanism. In this section we’ll have a closer look at this mechanism.

Assume we ask the compiler to compile the following \texttt{main} function:

```cpp
int main()
{
    process(3, 3);
}
```

Furthermore assume that the compiler has encountered the following function declarations when it’s about to compile \texttt{main}:

```cpp
template <typename T>
void process(T &t1, int i); // 1

template <typename T1, typename T2>
void process(T1 const &t1, T2 const &t2); // 2

template <typename T>
void process(T const &t, double d); // 3
```
template <typename T>
void process(T const &t, int i); // 4

template <>
void process<int, int>(int i1, int i2); // 5
void process(int i1, int i2); // 6
void process(int i, double d); // 7
void process(double d, int i); // 8
void process(double d1, double d2); // 9
void process(std::string s, int i) // 10
int add(int, int); // 11

The compiler, having read main’s statement, must now decide which function must actually be called. It proceeds as follows:

• First, a set of candidate functions is constructed. This set contains all functions that:
  - are visible at the point of the call;
  - have the same names as the called function.

As function 11 has a different name, it is removed from the set. The compiler is left with a set of 10 candidate functions.

• Second, the set of viable functions is constructed. Viable functions are functions for which type conversions exist that can be applied so as to match the types of the function's parameters to the types of the actual arguments.

This implies that at least the number of arguments must match the number of parameters of the viable functions. Function 10’s first argument is a string. As a string cannot be initialized by an int value no appropriate conversion exists and function 10 is removed from the list of candidate functions. double parameters can be retained. Standard conversions do exists for ints to doubles, so all functions having ordinary double parameters can be retained. Therefore, the set of viable functions consists of functions 1 through 9.

At this point the compiler tries to determine the types of the template type parameters. This step is outlined in the following subsection.

21.15 Determining the template type parameters

Having determined the set of candidate functions and from that set the set of viable functions the compiler must now determine the actual types of the template type parameters.

It may use any of the three standard template parameter transformation procedures (cf. section 21.4) when trying to match actual types to template type parameters. In this process it concludes that no type can be determined for the T in function 1’s T &t1 parameter as the argument 3 is a constant int value. Thus function 1 is removed from the list of viable functions. The compiler is
now confronted with the following set of potentially instantiated function templates and ordinary
functions:

```plaintext
void process(T1 [= int] const &t1, T2 [= int] const &t2); // 2
void process(T [= int] const &t, double d); // 3
void process(T [= int] const &t, int i); // 4
void process<int, int>(int i1, int i2); // 5
void process(int i1, int i2); // 6
void process(int i, double d); // 7
void process(double d, int i); // 8
void process(double d1, double d2); // 9
```

The compiler associates a direct match count value to each of the viable functions. The direct match
count counts the number of arguments that can be matched to function parameters without an
(automatic) type conversion. E.g., for function 2 this count equals 2, for function 7 it is 1 and for
function 9 it is 0. The functions are now (decrementally) sorted by their direct match count values:

<table>
<thead>
<tr>
<th>Function</th>
<th>Match Count</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void process(T1 [= int] const &amp;t1, T2 [= int] const &amp;t2);</code></td>
<td>2</td>
</tr>
<tr>
<td><code>void process(T [= int] const &amp;t, int i);</code></td>
<td>2</td>
</tr>
<tr>
<td><code>void process&lt;T [= int] const &amp;t, double d&gt;;</code></td>
<td>2</td>
</tr>
<tr>
<td><code>void process&lt;int, int&gt;(int i1, int i2);</code></td>
<td>2</td>
</tr>
<tr>
<td><code>void process(int i1, int i2);</code></td>
<td>2</td>
</tr>
<tr>
<td><code>void process(int i, double d);</code></td>
<td>1</td>
</tr>
<tr>
<td><code>void process(double d, int i);</code></td>
<td>1</td>
</tr>
<tr>
<td><code>void process(double d1, double d2);</code></td>
<td>0</td>
</tr>
</tbody>
</table>

If there is no draw for the top value the corresponding function is selected and the function selection
process is completed.

When multiple functions appear at the top the compiler verifies that no ambiguity has been encoun-
tered. An ambiguity is encountered if the sequences of parameters for which type conversions were
(not) required differ. As an example consider functions 3 and 8. Using D for ‘direct match’ and C
for ‘conversion’ the arguments match function 3 as D,C and function 8 as C,D. Assuming that 2, 4,
5 and 6 were not available, then the compiler would have reported an ambiguity as the sequences
of argument/parameter matching procedures differ for functions 3 and 8. The same difference is en-
countered comparing functions 7 and 8, but no such difference is encountered comparing functions
3 and 7.

At this point there is a draw for the top value and the compiler proceeds with the subset of associated
functions (functions 2, 4, 5 and 6). With each of these functions an ‘ordinary parameter count’ is
associated counting the number of non-template parameters of the functions. The functions are
decrementally sorted by this count, resulting in:

<table>
<thead>
<tr>
<th>Function</th>
<th>Ordin. Param. Count</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void process(int i1, int i2);</code></td>
<td>2 // 6</td>
</tr>
<tr>
<td><code>void process(T [= int] const &amp;t, int i);</code></td>
<td>1 // 4</td>
</tr>
<tr>
<td><code>void process&lt;T1 [= int] const &amp;t1, T2 [= int] const &amp;t2&gt;;</code></td>
<td>0 // 2</td>
</tr>
<tr>
<td><code>void process&lt;int, int&gt;(int i1, int i2);</code></td>
<td>0 // 5</td>
</tr>
</tbody>
</table>

Now there is no draw for the top value. The corresponding function (function 6) is selected and the function selection process is completed. Function 6 is used in `main`’s function
call statement.
Had function 6 not been defined, function 4 would have been used. Assuming that neither function 4 nor function 6 had been defined, the selection process would continue with functions 2 and 5:

<table>
<thead>
<tr>
<th>ordin. param.</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>void process(T1 [= int] const &amp;t1, T2 [= int] const &amp;t2);</td>
<td>0 // 2</td>
</tr>
<tr>
<td>void process&lt;int, int&gt;(int i1, int i2);</td>
<td>0 // 5</td>
</tr>
</tbody>
</table>

In this situation a draw is encountered once again and the selection process continues. A ‘type of function’ value is associated with each of the functions having the highest ordinary parameter count and these functions are decrementally sorted by their type of function values. Value 2 is associated to ordinary functions, value 1 to template explicit specializations and value 0 to plain function templates.

If there is no draw for the top value the corresponding function is selected and the function selection process is completed. If there is a draw the compiler reports an ambiguity and cannot determine which function to call. Assuming only functions 2 and 5 existed then this selection step would have resulted in the following ordering:

<table>
<thead>
<tr>
<th>function type</th>
</tr>
</thead>
<tbody>
<tr>
<td>void process&lt;int, int&gt;(int i1, int i2);</td>
</tr>
<tr>
<td>void process(T1 [= int] const &amp;t1, T2 [= int] const &amp;t2);</td>
</tr>
</tbody>
</table>

Function 5, the template explicit specialization, would have been selected. Here is a summary of the function template selection mechanism (cf. figure Figure 21.1):

- The set of candidate functions is constructed: identical names;
- The set of viable functions is constructed: correct number of parameters and available type conversions;
- Template type determination, dropping templates whose type parameters cannot be determined;
- Decrementally sort the functions by their direct match count values. If there is no draw for the top value the associated function is selected, completing the selection process.
- Inspect the functions associated with the top value for ambiguities in automatic type conversion sequences. If different sequences are encountered report an ambiguity and terminate the selection process.
- Decrementally sort the functions associated with the top value by their ordinary parameter count values. If there is no draw for the top value the associated function is selected, completing the selection process.
- Decrementally sort the functions associated with the top value by their function type values using 2 for ordinary functions, 1 for template explicit specializations and 0 for plain function templates. If there is no draw for the top value the associated function is selected, completing the selection process.
- Report an ambiguity and terminate the selection process.
CHAPTER 21. FUNCTION TEMPLATES

Create candidate functions (names match) → Create viable functions (#params match) → Determine template parameters

Sort by direct-match count → Type conversion ambiguity check → Sort by # ordinary params.

Sort by function type → Select the topmost function

Report Ambiguity

Figure 21.1: The function template selection mechanism
21.16  SFINAE: Substitution Failure Is Not An Error

Consider the following struct definition:

```c
struct Int
{
    typedef int type;
};
```

Although at this point it may seem strange to embed a typedef in a struct, in chapter 23 we'll encounter situations where this is actually very useful. It allows us to define a variable of a type that is required by the template. E.g., (ignore the use of typename in the following function parameter list, but see section 22.2.1 for details):

```c
template <typename Type>
void func(typename Type::type value)
{
}
```

To call `func(10)` `Int` has to be specified explicitly since there may be many structs that define `type` the compiler needs some assistance. The correct call is `func<Int>(10)`. Now that it's clear that `Int` is meant, and the compiler correctly deduces that `value` is an `int`.

But templates may be overloaded and our next definition is:

```c
template <typename Type>
void func(Type value)
{
}
```

Now, to make sure this function is used we specify `func<int>(10)` and again this compiles flawlessly.

But as we've seen in the previous section when the compiler determines which template to instantiate it creates a list of viable functions by matching the parameter types of available function prototypes with the provided actual argument types. To do so it has to determine the types of the parameters and herein lies a problem.

When evaluating `Type = int` the compiler encounters the prototypes `func<int::type>` (first template definition) and `func<int>` (second template definition). But there is no `int::type`, and so in a way this generates an error. However, the error results from substituting the provided template type argument into the various template definitions.

A type-problem caused by substituting a type in a template definition is not considered an error, but merely an indication that that particular type cannot be used in that particular template. The template is therefore removed from the list of candidate functions.

This principle is known as substitution failure is not an error (SFINAE) and it is often used by the compiler to select not only a simple overloaded function (as shown here) but also to choose among available template specializations (see also chapters 22 and 23).
21.17 Summary of the template declaration syntax

In this section the basic syntactic constructions for declaring templates are summarized. When defining templates, the terminating semicolon should be replaced by a function body.

Not every template declaration may be converted into a template definition. If a definition may be provided it is explicitly mentioned.

- A plain template declaration (a definition may be provided):

  template <typename Type1, typename Type2>
  void function(Type1 const &t1, Type2 const &t2);

- A template instantiation declaration (no definition may be provided):

  template
  void function<int, double>(int const &t1, double const &t2);

- A template using explicit types (no definition may be provided):

  void (*fp)(double, double) = function<double, double>;
  void (*fp)(int, int) = function<int, int>;

- A template explicit specialization (a definition may be provided):

  template <>
  void function<char *, char *>(char *const &t1, char *const &t2);

- A template declaration declaring friend function templates within class templates (covered in section 22.10, no definition may be provided):

  friend void function<Type1, Type2>(parameters);
Chapter 22

Class Templates

Templates can not only be constructed for functions but also for complete classes. Consider constructing a class template when a class should be able to handle different types of data. Class templates are frequently used in C++: chapter 12 discusses data structures like vector, stack and queue, that are implemented as class templates. With class templates, the algorithms and the data on which the algorithms operate are completely separated from each other. To use a particular data structure in combination with a particular data type only the data type needs to be specified when defining or declaring a class template object (as in stack<int> iStack).

In this chapter constructing and using class templates is discussed. In a sense, class templates compete with object oriented programming (cf. chapter 14), that uses a mechanism resembling that of templates. Polymorphism allows the programmer to postpone the implementation of algorithms by deriving classes from base classes in which algorithms are only partially implemented. The actual definition and processing of the data upon which the algorithms operate may be postponed until derived classes are defined. Likewise, templates allow the programmer to postpone the specification of the data upon which the algorithms operate. This is most clearly seen with abstract containers, which completely specify the algorithms and at the same time leave the data type on which the algorithms operate completely unspecified.

There exists an intriguing correspondence between the kind of polymorphism we’ve encountered in chapter 14 and certain uses of class templates. In their book C++ Coding Standards (Addison-Wesley, 2005) Sutter and Alexandrescu refer to static polymorphism and dynamic polymorphism. Dynamic polymorphism is what we use when overriding virtual members. Using vtables the function that is actually called depends on the type of object a (base) class pointer points at. Static polymorphism is encountered in the context of templates, and is discussed and compared to dynamic polymorphism in section 22.12.

Generally, class templates are easier to use than polymorphism. It is certainly easier to write stack<int> iStack to create a stack of ints than to derive a new class Istack: public stack and to implement all necessary member functions defining a similar stack of ints using object oriented programming. On the other hand, for each different type for which an object of a class template is defined another, possibly complete class must be reinstantiated. This is not required in the context of object oriented programming where derived classes use, rather than copy, the functions that are already available in their base classes (but see also section 22.11).

Previously we’ve already used class templates. Objects like vector<int> vi and vector<string> vs are commonly used. The data types for which these templates are defined and instantiated are an inherent part of such container types. It is stressed that it is the combination of a class template type and its template parameter(s), rather than the mere class template’s
type that defines or generates a type. So \texttt{vector<int> is a type as is vector<string>}. Such types could very well be represented by typedefs:

\begin{verbatim}
typedef std::vector<int> IntVector;
typedef std::vector<std::string> StringVector;
\end{verbatim}

\begin{verbatim}
IntVector vi;
StringVector vs;
\end{verbatim}

\section*{22.1 Defining class templates}

Having covered the construction of function templates, we're now ready for the next step: constructing class templates. Many useful class templates already exist. Rather than illustrating the construction of a class template by looking at an already existing class template the construction of another potentially useful new class template will be undertaken.

The new class implements a \textit{circular queue}. A circular queue has a fixed number of \texttt{max_size} elements. New elements are inserted at its back and only its head and tail elements can be accessed. Only the head element can be removed from a circular queue. Once \texttt{n} elements have been appended the next element is inserted again at the queue's (physical) first position. The circular queue allows insertions until it holds \texttt{max_size} elements. As long as a circular queue contains at least one element elements may be removed from it. Trying to remove an element from an empty circular queue or to add another element to a full circular queue results in exceptions being thrown. In addition to other constructors a circular queue must offer a constructor initializing its objects for \texttt{max_size} elements. This constructor must make available the memory for the \texttt{max_size} elements but must not call those elements default constructors (hinting at the use of the placement new operator). A circular queue should offer value semantics as well as a move constructor.

Please note that in the above description the actual data type that is used for the circular queue is nowhere mentioned. This is a clear indication that our class could very well be defined as a \texttt{class template}. Alternatively, the class could be defined for some concrete data type which is then abstracted when converting the class to a class template.

The actual construction of a class template is provided in the next section, where the class template \texttt{CirQue} (circular queue) is developed.

\subsection*{22.1.1 Constructing the circular queue: CirQue}

The construction of a class template is illustrated in this section. Here, we'll develop the class template \texttt{CirQue} (circular queue). This class template has one template type parameter, \texttt{Data}, representing the data type that is stored in the circular queue. The outline of the interface of this class template looks like this:

\begin{verbatim}
template<typename Data>
class CirQue
{
    // member declarations
};
\end{verbatim}

A class template's definition starts like a function template's definition:
22.1. DEFINING CLASS TEMPLATES

- The keyword **template**, starting a template definition or declaration.
- The angle bracket enclosed list following **template**: a list containing one or more comma-separated elements is called the **template parameter list**. Template parameter lists may have multiple elements, like this:

  ```
  typename Type1, typename Type2, typename Type3
  ```

When a class template defines multiple template type parameters they are matched in sequence with the list of template type arguments provided when defining objects of such a class template. Example:

```
template <typename Type1, typename Type2, typename Type3>
class MultiTypes
{
    ...
};
MultiTypes<int, double, std::string> multiType;
// Type1 is int, Type2 is double, Type3 is std::string
```

- Inside the template parameter list we find the **formal type** name (Data for CirQue). It is a formal (type) name, like the formal types used in function template parameter lists.

- Following the template header the class interface is defined. It may use the formal type names defined in the template header as type names.

Once the CirQue class template has been defined it can be used to create all kinds of circular queues. As one of its constructors expects a size_t argument defining the maximum number of elements that can be stored in the circular queue, circular queues could be defined like this:

```
CirQue<int> cqi(10);     // max 10 ints
CirQue<std::string> cqstr(30);  // max 30 strings
```

As noted in the introductory section of this chapter the combination of name of the class template and the data type for which it is instantiated defines a data type. Also note the similarity between defining a std::vector (of some data type) and a CirQue (of some data type).

Like std::map containers class templates may be defined with multiple template type parameters.

Back to CirQue. A CirQue must be capable of storing max_size Data elements. These elements are eventually stored in memory pointed at by a pointer Data *d_data, initially pointing to raw memory. New elements are added at the backside of the CirQue. A pointer Data *d_back is used to point to the location where the next element is going to be stored. Likewise, Data *d_front points to the location of the CirQue’s first element. Two size_t data members are used to monitor the filling state of the CirQue: d_size represents the number of elements currently stored in the CirQue, d_maxSize represents the maximum number of elements that the CirQue can contain. Thus, the CirQue’s data members are:

```
size_t d_size;
size_t d_maxSize;
Data *d_data;
Data *d_front;
Data *d_back;
```
22.1.2 Non-type parameters

Function template parameters are either template type parameters or template non-type parameters (actually, a third type of template parameter exists, the \textit{template template parameter}, which is discussed in chapter \textit{23} (section \textit{23.4}).

Class templates may also define non-type parameters. Like the function template non-type parameters they must be (integral) constants whose values must be known at object instantiation time.

Different from function template non-type parameters the values of class template non-type parameters are \textit{not} deduced by the compiler using arguments passed to class template members.

Assume we extend our design of the class template \textit{CirQue} so that it defines a second (non-type) parameter \texttt{size_t Size}. Our intent is to use this \texttt{Size} parameter in the constructor defining an array parameter of \texttt{Size} elements of type \texttt{Data}.

The \textit{CirQue} class template now becomes (only showing the relevant constructor):

\begin{verbatim}
template <typename Data, size_t Size>
class CirQue
{
    // ... data members
    public:
        CirQue(Data const (&arr)[Size]);
        ...
};

template <typename Data, size_t Size>
CirQue<Data, Size>::CirQue(Data const (&arr)[Size])
    :
        d_maxSize(Size),
        d_size(0),
        d_data(operator new(Size* sizeof(Data))),
        d_front(d_data),
        d_back(d_data),
    {
        std::copy(arr, arr + Size, back_inserter(*this));
    }
\end{verbatim}

Unfortunately, this setup doesn’t satisfy our needs as the values of template non-type parameters are not deduced by the compiler. When the compiler is asked to compile the following \texttt{main} function it reports a mismatch between the required and actual number of template parameters:

\begin{verbatim}
int main()
{
    int arr[30];

    CirQue<int> ap(arr);
}
/*
 Error reported by the compiler:

 In function ‘int main()’:
   error: wrong number of template arguments (1, should be 2)
   error: provided for ‘template<class Data, size_t Size>
\end{verbatim}
22.1. DEFINING CLASS TEMPLATES

class CirQue'

* /

Defining Size as a non-type parameter having a default value doesn't work either. The compiler always uses the default unless its value is explicitly specified. Reasoning that Size can be 0 unless we need another value, we might be tempted to specify size_t Size = 0 in the template's parameter type list. Doing so we create a mismatch between the default value 0 and the actual size of the array arr as defined in the above main function. The compiler, using the default value, reports:

```cpp
In instantiation of 'CirQue<int, 0>':
...
error: creating array with size zero ('0')
```

So, although class templates may use non-type parameters they must always be specified like type parameters when an object of that class is defined. Default values can be specified for those non-type parameters causing the compiler to use the default when the non-type parameter is left unspecified.

Default template parameter values (either type or non-type template parameters) may not be specified when defining template member functions. In general: function template definitions (and thus: class template member functions) may not be given default template (non) type arguments. If default template arguments are to be used for class template members, they have to be specified by the class interface.

Similar to non-type parameters of function templates default argument values for non-type class template parameters may only be specified as constants:

- Global variables have constant addresses, which can be used as arguments for non-type parameters.
- Local and dynamically allocated variables have addresses that are not known by the compiler when the source file is compiled. These addresses can therefore not be used as arguments for non-type parameters.
- Lvalue transformations are allowed: if a pointer is defined as a non-type parameter, an array name may be specified.
- Qualification conversions are allowed: a pointer to a non-const object may be used with a non-type parameter defined as a const pointer.
- Promotions are allowed: a constant of a 'narrower' data type may be used when specifying a default argument for a non-type parameter of a 'wider' type (e.g., a short can be used when an int is called for, a long when a double is called for).
- Integral conversions are allowed: if a size_t parameter is specified, an int may be used as well.
- Variables cannot be used to specify template non-type parameters as their values are not constant expressions. Variables defined using the const modifier, however, may be used as their values never change.

Although our attempts to define a constructor of the class CirQue accepting an array as its argument has failed so far, we're not yet out of options. In the next section a method is described that does allow us to reach our goal.
22.1.3 Member templates

Our previous attempt to define a template non-type parameter that is initialized by the compiler to the number of elements of an array failed because the template's parameters are not implicitly deduced when a constructor is called. Instead, they must explicitly be specified when an object of the class template is defined. As the template arguments are specified just before the template's constructor is called the compiler doesn't have to deduce anything and it can simply use the explicitly specified template arguments.

In contrast, when function templates are used, the actual template parameters are deduced from the arguments used when calling the function. This opens up an alley leading to the solution of our problem. If the constructor itself is turned into a function template (having its own template header), then the compiler will be able to deduce the non-type parameter's value and there is no need anymore to specify it explicitly using a class template non-type parameter.

Members (functions or nested classes) of class templates that are themselves templates are called member templates.

Member templates are defined as any other template, including its own template header.

When converting our earlier CirQue(Data const (&array)[Size]) constructor into a member template the class template's Data type parameter can still be used, but we must provide the member template with a non-type parameter of its own. Its declaration in the (partially shown) class interface looks like this:

```cpp
template <typename Data>
class CirQue
{
    public:
        template <size_t Size>
        explicit CirQue(Data const (&arr)[Size]);
};
```

Its implementation becomes:

```cpp
template <typename Data>
template <size_t Size>
CirQue<Data>::CirQue(Data const (&arr)[Size])
:
    d_size(0),
    d_maxSize(Size),
    d_data(static_cast<Data*>(operator new(sizeof(arr)))),
    d_front(d_data),
    d_back(d_data)
{
    std::copy(arr, arr + Size, back_inserter(*this));
}
```

The implementation uses the STL's `copy` algorithm and a `back_inserter` adapter to insert the array's elements into the CirQue. To use the `back_inserter` CirQue must define two (public) typedefs (cf. section 18.2.2):

```cpp
typedef Data value_type;
typedef value_type const &const_reference;
```
22.1. DEFINING CLASS TEMPLATES

Member templates have the following characteristics:

- Two template headers must be used: the class template’s template header is specified first followed by the member template’s template header;
- Normal access rules apply: the member template can be used by programs to construct an `CirQue` object of a given data type. As usual for class templates, the data type must be specified when the object is constructed. To construct a `CirQue` object from the array `int array[30]` we define:

  ```cpp
  CirQue<int> object(array);
  ```
- Any member can be defined as a member template, not just a constructor;
- When a template member is implemented below its class interface, the template class header must precede the function template header of the member template;
- The implementation of the member template must specify its proper scope. The member template is defined as a member of the class `CirQue`, instantiated for the formal template parameter type `Data`;
- The template parameter names in the declaration and implementation must be identical;
- The member template should be defined inside its proper namespace environment. The organization of files defining class templates within a namespace should therefore be:

```cpp
namespace SomeName
{
    template <typename Type, ...> // class template definition
    class ClassName
    {
        ...
    };

    template <typename Type, ...> // non-inline member definition(s)
    ClassName<Type, ...>::member(...)
    {
        ...
    } // namespace closes
```

A potentially occurring problem remains. Assume that in addition to the above member template a `CirQue<Data>::CirQue(Data const *data)` has been defined. Some (here not further elaborated) protocol could be defined allowing the constructor to determine the number of elements that should be stored in the `CirQue` object. When we now define

```cpp
CirQue<int> object(array);
```

it is this latter constructor, rather than the member template, that the compiler will use.

The compiler selects this latter constructor as it is a more specialized version of a constructor of the class `CirQue` than the member template (cf. section 21.14). Problems like these can be solved by defining the constructor `CirQue(Data const *data)` into a member template as well or by defining a constructor using two parameters, the second parameter defining the number of elements to copy.
When using the former constructor (i.e., the member template) it must define a template type parameter \texttt{Data2}. Here ‘Data’ cannot be used as template parameters of a member template may not shadow template parameters of its class. Using \texttt{Data2} instead of \texttt{Data} takes care of this subtlety. The following declaration of the constructor \texttt{CirQue(Data2 \texttt{const *} )} could appear in \texttt{CirQue}'s header file:

```cpp
template <typename Data2>
class CirQue
{
    template <typename Data2>
    explicit CirQue(Data2 \texttt{const *} \texttt{data});
};
```

Here is how the two constructors are selected in code defining two \texttt{CirQue} objects:

```cpp
int main()
{
    int array[30];
    int *iPtr = array;

    CirQue<int> ac(array); // calls \texttt{CirQue(Data \texttt{const \&arr})[\textbf{Size}]}
    CirQue<int> acPtr(iPtr); // calls \texttt{CirQue(Data2 \texttt{const *} )}
}
```

### 22.1.4 \texttt{CirQue}'s constructors and member functions

It’s time to return to \texttt{CirQue}'s design and construction again.

The class \texttt{CirQue} offers various member functions. Normal design principles should be adhered to when constructing class template members. Class template type parameters should preferably be defined as \texttt{Type \texttt{const \&}}, rather than \texttt{Type}, to prevent unnecessary copying of large data structures. Template class constructors should use member initializers rather than member assignment within the body of the constructors. Member function definitions should preferably not be provided in-class but below the class interface. Since class template member functions are function templates their definitions should be provided in the header file offering the class interface. They \textit{may} be given the \texttt{\texttt{inline}} attribute.

\texttt{CirQue} declares several constructors and (public) members (their definitions are provided as well; all definitions are provided below the class interface).

Here are the constructors and the destructor:

- \texttt{explicit \texttt{CirQue(size\_t \texttt{maxSize} = 0)}:}
  
    Constructor initializing a \texttt{CirQue} capable of storing \texttt{max\_size} \texttt{Data} elements. As the constructor’s parameter is given a default argument value this constructor can also be used as a default constructor, allowing us to define, e.g., vectors of \texttt{CirQue}s. The constructor initializes the \texttt{CirQue} object’s \texttt{d\_data} member to a block of raw memory and \texttt{d\_front} and \texttt{d\_back} are initialized to \texttt{d\_data}. As class template member functions are themselves function templates their implementations outside of the class template’s interface must start with the class template’s template header. Here is the implementation of the \texttt{CirQue(size\_t)} constructor:

```cpp
template<typename Data>
```
22.1. DEFINING CLASS TEMPLATES

CirQue<Data>::CirQue(size_t maxSize):
    d_size(0),
    d_maxSize(maxSize),
    d_data(
        maxSize == 0 ?
        0 :
        static_cast<Data *>(
            operator new(maxSize * sizeof(Data)))
    ),
    d_front(d_data),
    d_back(d_data)
{}  

• CirQue(CirQue<Data> const &other):

The copy constructor has no special features. It uses a private support member inc to
increment d_back (see below) and placement new to copy the other’s Data elements
to the current object. The implementation of the copy constructor is straightforward:

template<typename Data>
CirQue<Data>::CirQue(CirQue<Data> const &other):
    d_size(other.d_size),
    d_maxSize(other.d_maxSize),
    d_data(
        d_maxSize == 0 ?
        0 :
        static_cast<Data *>(
            operator new(d_maxSize * sizeof(Data)))
    ),
    d_front(d_data + (other.d_front - other.d_data))
{
    Data const *src = other.d_front;
    d_back = d_front;
    for (size_t count = 0; count != d_size; ++count)
    {
        new(d_back) Data(*src);
        d_back = inc(d_back);
        if (++src == other.d_data + d_maxSize)
            src = other.d_data;
    }
}

• CirQue(CirQue<Data> &&tmp):

The move constructor merely initializes the current object’s d_data pointer to 0 and
swaps (see the member swap, below) the temporary object with the current object. 
CirQue’s destructor inspects d_data and immediately returns when it’s zero. Imple-
mentation:

template<typename Data>
CirQue<Data>::CirQue(CirQue<Data> &&tmp):
    d_data(0)
CHAPTER 22. CLASS TEMPLATES

{  
    swap(tmp);  
}

- CirQue(CirQue(Data const (&arr)[Size])):

  This constructor is declared as a member template, providing the Size non-type parameter. It allocates room for Size data elements and copies arr's contents to the newly allocated memory. Implementation:

  template <typename Data>
  template <size_t Size>
  CirQue<Data>::CirQue(Data const (&arr)[Size]):
    d_size(0),
    d_maxSize(Size),
    d_data(static_cast<Data*>(operator new(sizeof(arr)))),
    d_front(d_data),
    d_back(d_data)
  {
    std::copy(arr, arr + Size, back_inserter(*this));
  }

- CirQue(CirQue(Data const *data, size_t size)):

  This constructor acts very much like the previous one, but is provided with a pointer to the first Data element and with a size_t providing the number of elements to copy. In our current design the member template variant of this constructor is left out of the design. As the implementation of this constructor is very similar to that of the previous constructor, it is left as an exercise to the reader.

- ~CirQue():

  The destructor inspects the d_data member. If it is zero then nothing has been allocated and the destructor immediately returns. This may occur in two situations: the circular queue contains no elements or the information was grabbed from a temporary object by some move operation, setting the temporary's d_data member to zero. Otherwise d_size elements are destroyed by explicitly calling their destructors followed by returning the element's raw memory to the common pool. Implementation:

  template<typename Data>
  CirQue<Data>::~CirQue()
  {
    if (d_data == 0)
      return;
    for (; d_size--; )
    {
      d_front->~Data();
      d_front = inc(d_front);
    }
    operator delete(d_data);
  }

Here are Cirque’s members:

- CirQue &operator=(CirQue<Data> const &other):
The copy assignment operator has a standard implementation:

```cpp
template<typename Data>
CirQue<Data> &CirQue<Data>::operator=(CirQue<Data> const &rhs)
{
    CirQue<Data> tmp(rhs);
    swap(tmp);
    return *this;
}
```

The move assignment operator also has a standard implementation. As its implementation merely calls `swap` it is defined as an inline function template:

```cpp
inline CirQue<Data> &CirQue<Data>::operator=(CirQue<Data> &&tmp)
{
    swap(tmp);
    return *this;
}
```

- `void pop_front()`: removes the element pointed at by `d_front` from the `CirQue`. Throws an exception if the `CirQue` is empty. The exception is thrown as a `CirQue<Data>::EMPTY` value, defined by the enum `CirQue<Data>::Exception` (see `push_back`). The implementation is straightforward (explicitly calling the destructor of the element that is removed):

```cpp
template<typename Data>
void CirQue<Data>::pop_front()
{
    if (d_size == 0)
        throw EMPTY;
    d_front->~Data();
    d_front = inc(d_front);
    --d_size;
}
```

- `void push_back(Data const &object)`: adds another element to the `CirQue`. Throws a `CirQue<Data>::FULL` exception if the `CirQue` is full. The exceptions that can be thrown by a `CirQue` are defined in its `Exception` enum:

```cpp
enum Exception
{
    EMPTY,
    FULL
};
```

A copy of `object` is installed in the `CirQue`'s raw memory using placement `new` and its `d_size` is incremented.

```cpp
template<typename Data>
void CirQue<Data>::push_back(Data const &object)
{
    if (d_size == d_maxSize)
```
throw FULL;
    new (d_back) Data(object);
    d_back = inc(d_back);
    ++d_size;
}

- void swap(CirQue<
    Data> &
other):

  swaps the current CirQue object with another CirQue<
    Data> object;

    template<typename Data>
    void CirQue<Data>::swap(CirQue<Data> &other)
    {
      char tmp[sizeof(CirQue<Data>)];
      memcpy(tmp, &other, sizeof(CirQue<Data>));
      memcpy(&other, this, sizeof(CirQue<Data>));
      memcpy(this, tmp, sizeof(CirQue<Data>));
    }

The remaining public members all consist of one-liners and were implemented as inline function
templates:

- Data &back():

  returns a reference to the element pointed at by d_back (undefined result if the CirQue is empty):

    template<typename Data>
    inline Data &CirQue<Data>::back()
    {
      return d_back == d_data ? d_data[d_maxSize - 1] : d_back[-1];
    }

- Data &front():

  returns reference to the element pointed at by d_front (undefined result if the CirQue is empty):

    template<typename Data>
    inline Data &CirQue<Data>::front()
    {
      return *d_front;
    }

- bool empty() const:

  returns true if the CirQue is empty;

    template<typename Data>
    inline bool CirQue<Data>::empty() const
    {
      return d_size == 0;
    }

- bool full() const:

  returns true if the CirQue is full;

    template<typename Data>
22.1. DEFINING CLASS TEMPLATES

```cpp
inline bool CirQue<Data>::full() const
{
    return d_size == d_maxSize;
}

• size_t size() const:
    returns the number of elements currently stored in the CirQue;
    template<typename Data>
    inline size_t CirQue<Data>::size() const
    {
        return d_size;
    }

• size_t maxSize() const:
    returns the maximum number of elements that can be stored in the CirQue;
    template<typename Data>
    inline size_t CirQue<Data>::maxSize() const
    {
        return d_maxSize;
    }
```

Finally, the class has one private member, inc, returning a cyclically incremented pointer into CirQue's raw memory:

```cpp
template<typename Data>
Data *CirQue<Data>::inc(Data *ptr)
{
    ++ptr;
    return ptr == d_data + d_maxSize ? d_data : ptr;
}
```

22.1.5 Using CirQue objects

When objects of a class template are instantiated, only the definitions of all the template's member functions that are actually used must have been seen by the compiler.

That characteristic of templates could be refined to the point where each definition is stored in a separate function template definition file. In that case only the definitions of the function templates that are actually needed would have to be included. However, it is hardly ever done that way. Instead, the usual way to define class templates is to define the interface and to define the remaining function templates immediately below the class template's interface (defining some functions inline).

Now that the class CirQue has been defined, it can be used. To use the class its object must be instantiated for a particular data type. In the following example it is initialized for data type std::string:

```cpp
#include "cirque.h"
#include <iostream>
#include <string>
using namespace std;
```
int main()
{
    CirQue<string> ci(4);
    ci.push_back("1");
    ci.push_back("2");
    cout << ci.size() << ' ' << ci.front() << ' ' << ci.back() << '
';

    ci.push_back("3");
    ci.pop_front();
    ci.push_back("4");
    ci.pop_front();
    ci.push_back("5");
    cout << ci.size() << ' ' << ci.front() << ' ' << ci.back() << '
';

    CirQue<string> copy(ci);
    copy.pop_front();
    cout << copy.size() << ' ' << copy.front() << ' ' << copy.back() << '
';

    int arr[] = {1, 3, 5, 7, 9};
    CirQue<int> ca(arr);
    cout << ca.size() << ' ' << ca.front() << ' ' << ca.back() << '
';

    // int *ap = arr;
    // CirQue<int> cap(ap);
}

This program produces the following output:

    2 1 2
    3 3 5
    2 4 5
    5 1 9

22.1.6 Default class template parameters

Different from function templates, template parameters of template classes may be given default argument values. This holds true for both template type- and template non-type parameters. If default template arguments were defined and if a class template object is instantiated without specifying arguments for its template parameters then the template parameter's defaults are used.

When defining defaults keep in mind that they should be suitable for the majority of instantiations of the class. E.g., for the class template `CirQue` the template's type parameter list could have been altered by specifying `int` as its default type:

    template <typename Data = int>

Even though default arguments can be specified, the compiler must still be informed that object definitions refer to templates. When instantiating class template objects using default template arguments the type specifications may be omitted but the angle brackets must be retained. Assuming a default type for the `CirQue` class, an object of that class may be defined as:

    CirQue<> intCirQue(10);
22.1. DEFINING CLASS TEMPLATES

Default template arguments cannot be specified when defining template members. So, the definition of, e.g., the `push_back` member must always begin with the same template specification:

```
template <typename Data>
```

When a class template uses multiple template parameters, all may be given default values. Like default function arguments, once a default value is used all remaining template parameters must also use their default values. A template type specification list may not start with a comma, nor may it contain multiple consecutive commas.

22.1.7 Declaring class templates

Class templates may also be declared. This may be useful in situations where forward class declarations are required. To declare a class template, simply remove its interface (the part between the curly braces):

```
template <typename Data>
class CircQue;
```

Default template arguments may also be specified when declaring class templates. However, default template arguments cannot be specified for both the declaration and the definition of a class template. As a rule of thumb default template arguments should be omitted from declarations, as class template declarations are never used when instantiating objects but are only occasionally used as forward references. Note that this differs from default parameter value specifications for member functions in ordinary classes. Such defaults are always specified when declaring the member functions in the class interface.

22.1.8 Preventing template instantiations

In C++ templates are instantiated when the address of a function template or class template object is taken or when a function template or class template is used. As described in section 22.1.7 it is possible to (forward) declare a class template to allow the definition of a pointer or reference to that template class or to allow it being used as a return type.

In other situations templates are instantiated when they are being used. If this happens many times (i.e., in many different source files) then this may slow down the compilation process considerably. The C++11 standard allows programmers to prevent templates from being instantiated. For this the C++11 standard introduces the `extern template` syntax. Example:

```
extern template class std::vector<int>;
```

Having declared the class template it can be used in its translation unit. E.g., the following function properly compiles:

```
#include <vector>
#include <iostream>
using namespace std;

extern template class vector<int>;
```

```
void vectorUser()
{
  vector<int> vi;
  cout << vi.size() << '\n';
}

But be careful:

- The declaration by itself does not make the class definition available. The `vector` header file still needs to be included to make the features of the class `vector` known to the compiler. But due to the `extern template` declaration none of the used members will be instantiated for the current compilation unit;

- The compiler assumes (as it always does) that what is declared has been implemented elsewhere. In this case the compiler encounters an implicit declaration: the features of the vector class that are actually used by the above program are not individually declared but they are declared as a group, using the `extern template` declaration. This not only holds true for explicitly used members but hidden members (copy constructors, move constructors, conversion operators, constructors called during promotions, to name a few): all are assumed by the compiler to have been instantiated elsewhere;

- Although the above source file compiles, the instantiations of the templates must be available before the linker can build the final program. To accomplish this one or more sourcefiles may be constructed in which all required instantiations are made available.

In a stand-alone program one might postpone defining the required members and wait for the linker to complain about unresolved external references. These may then be used to create a series of instantiation declarations which are then linked to the program to satisfy the linker. Not a very simple task, though, as the declarations must strictly match the way the members are declared in the class interface. An easier approach is to define an instantiation source file in which all facilities that are used by the program are actually instantiated in a function that is never called by the program. By adding this instantiation function to the source file containing `main` we can be sure that all required members are instantiated as well. Here is an example of how this can be done:

```cpp
#include <vector>
#include <iostream>

extern void vectorUser();

int main()
{
  vectorUser();
}

// this part is never called. It is added to make sure all required
// features of declared templates will also be instantiated.

namespace
{
  void instantiator()
  {
    std::vector<int> vi;
    vi.size();
  }
}
• Last, but certainly not least: a fully matching instantiation declaration of a class template (e.g., for `std::vector<int>`) looks like this:

    template class std::vector<int>;

Adding this to a source file, however, will instantiate the full class, i.e., all its members are now instantiated. This may not what you want, as it might needlessly inflate your final executable.

• On the other hand, if it is known that the required template members have already been instantiated elsewhere, then an `extern template` declaration can be used to prevent member instantiations in the current compilation unit, which may speed up compilation. E.g.,

    // the compiler assumes that required members of  
    // `vector<int>` have already been instantiated elsewhere  
    extern template class std::vector<int>;

    int main()
    {
      std::vector<int> vi(5); // constructor and operator[]  
      ++vi[0]; // are NOT instantiated  
    }

22.2 Static data members

When static members are defined in class templates, they are defined for every new type for which the class template is instantiated. As they are static members, there will only be one member per type for which the class template is instantiated. For example, in a class like:

    template <typename Type>
    class TheClass
    {
      static int s_objectCounter;
    };

There will be one `TheClass<Type>::s_objectCounter` for each different `Type` specification. The following object definitions result in the instantiation of just one single static variable, shared among the two objects:

    TheClass<int> theClassOne;
    TheClass<int> theClassTwo;

Mentioning static members in interfaces does not mean these members are actually defined. They are only declared and must be defined separately. With static members of class templates this is no different. The definitions of static members are usually provided immediately following (i.e., below) the template class interface. For example, the static member `s_objectCounter`'s definition, positioned just below its class interface, looks like this:

    template <typename Type> // definition, following  
    int TheClass<Type>::s_objectCounter = 0; // the interface
Here \( s_{\text{objectCounter}} \) is an int and is thus independent of the template type parameter \( \text{Type} \). Multiple instantiations of \( s_{\text{objectCounter}} \) for identical \( \text{Type} \)s cause no problem, as the linker will remove all but one instantiation from the final executable (cf. section 21.5).

In list-like constructions, where a pointer to objects of the class itself is required, the template type parameter \( \text{Type} \) must be used when defining the static variable. Example:

```cpp
template <typename Type>
class TheClass
{
    static TheClass *s_objectPtr;
};

template <typename Type>
TheClass<Type> *TheClass<Type>::s_objectPtr = 0;
```

As usual, the definition can be read from the variable name back to the beginning of the definition: \( s_{\text{objectPtr}} \) of the class \( \text{TheClass<Type>} \) is a pointer to an object of \( \text{TheClass<Type>} \).

When a static variable of a template's type parameter's type is defined, it should of course not be given the initial value 0. The default constructor (e.g., Type()) is usually more appropriate. Example:

```cpp
template <typename Type> // s_type's definition
Type TheClass<Type>::s_type = Type();
```

### 22.2.1 Extended use of the keyword ‘typename’

Until now the keyword `typename` has been used to indicate a template type parameter. However, it is also used to disambiguate code inside templates. Consider the following function template:

```cpp
template <typename Type>
Type function(Type t)
{
    Type::Ambiguous *ptr;
    return t + *ptr;
}
```

When this code is processed by the compiler, it complains with an -at first sight puzzling- error message like:

```
4: error: 'ptr' was not declared in this scope
```

The error message is puzzling as it was the programmer’s intention to declare a pointer to a type `Ambiguous` defined within the class template `Type`. But the compiler, confronted with `Type::Ambiguous` may interpret the statement in various ways. Clearly it cannot inspect `Type` itself trying to uncover `Type`'s true nature as `Type` is a template type. Because of this `Type`'s actual definition isn't available yet.

The compiler is confronted with two possibilities: either `Type::Ambiguous` is a static member of the as yet mysterious template `Type`, or it is a subtype of `Type`. As the standard specifies that the compiler must assume the former, the statement
Type::Ambiguous *ptr;

is interpreted as a multiplication of the static member Type::Ambiguous and the (now undeclared) entity ptr. The reason for the error message should now be clear: in this context ptr is unknown.

To disambiguate code in which an identifier refers to a subtype of a template type parameter the keyword typename must be used. Accordingly, the above code is altered into:

```cpp
template <typename Type>
Type function(Type t)
{
    typename Type::Ambiguous *ptr;
    return t + *ptr;
}
```

Classes fairly often define subtypes. When such subtypes appear inside template definitions as subtypes of template type parameters the typename keyword must be used to identify them as subtypes. Example: a class template Handler defines a typename Container as its template type parameter. It also defines a data member storing the iterator returned by the container's begin member. In addition Handler offers a constructor accepting any container supporting a begin member. Handler's class interface could then look like this:

```cpp
template <typename Container>
class Handler
{
    Container::const_iterator d_it;

    public:
    Handler(Container const &container)
    :
        d_it(container.begin())
    {};
};
```

What did we have in mind when designing this class?

- The typename Container represents any container supporting iterators.
- The container presumably supports a member begin. The initialization d_it(container.begin()) clearly depends on the template's type parameter, so it's only checked for basic syntactic correctness.
- Likewise, the container presumably supports a subtype const_iterator, defined in the class Container.

The final consideration is an indication that typename is required. If this is omitted and a Handler is instantiated the compiler produces a peculiar compilation error:

```cpp
#include "handler.h"
#include <vector>
using namespace std;
```
int main()
{
    vector<int> vi;
    Handler<vector<int>> ph(vi);
}

Reported error:
handler.h:4: error: syntax error before ‘;’ token
*/

Clearly the line

Container::const_iterator d_it;

in the class Handler causes a problem. It is interpreted by the compiler as a static member instead of a subtype. The problem is solved using typename:

template <typename Container>
class Handler
{
    typename Container::const_iterator d_it;
    ...
};

An interesting illustration that the compiler indeed assumes X::a to be a member a of the class X is provided by the error message we get when we try to compile main using the following implementation of Handler’s constructor:

Handler(Container const &container)
:
    d_it(container.begin())
{
    size_t x = Container::ios_end;
}

Reported error:

error: ‘ios_end’ is not a member of type ‘std::vector<int, std::allocator<int>>’
*/

Now consider what happens if the function template introduced at the beginning of this section doesn’t return a Type value, but a Type::Ambiguous value. Again, a subtype of a template type is referred to, and typename must be used:

template <typename Type>
typeid Type::Ambiguous function(Type t)
{
    return t.ambiguous();
}
Using `typename` in the specification of a return type is further discussed in section 23.1.1.

Typenames can be embedded in `typedef`s. As is often the case, this reduces the complexities of declarations and definitions appearing elsewhere. In the next example the type `Iterator` is defined as a subtype of the template type `Container`. `Iterator` may now be used without requiring the use of the keyword `typename`:

```c++
template <typename Container>
class Handler
{
    typedef typename Container::const_iterator Iterator;

    Iterator d_it;
    ... }
```

### 22.3 Specializing class templates for deviating types

The class `CirQue` can be used for many different types. Their common characteristic is that they can simply be pointed at by the class's `d_data` member. But this is not always as simple as it looks. What if `Data` turns out to be a `vector<int>`? For such data types the vanilla `CirQue` implementation cannot be used and a specialization could be considered. When considering a specialization one should also consider inheritance. Often a class derived from the class template accepting the incompatible data structure as its argument but otherwise equal to the original class template can easily be designed. The developmental advantage of inheritance over specialization is clear: the inherited class inherits the members of its base class while the specialization inherits nothing. All members defined by the original class template must be implemented again by the class template's specialization.

The specialization considered here is a true specialization in that the data members and representation used by the specialization greatly differ from the original `CirQue` class template. Therefore all members defined by the original class template must be modified to fit the specialization's data organization.

Like function template specializations class template specializations start with a template header that may or may not have an empty template parameter list. If the template parameters are directly specialized by the specialization it remains empty (e.g., `CirQue`'s template type parameter `Data` is specialized for `char * data`). But the template parameter list may show `typename Data` when specializing for a `vector<Data>`, i.e., a vector storing any type of data. This leads to the following principle:

A template specialization is recognized by the template argument list following a function or class template's `name` and not by an empty template parameter list. Class template specializations may have non-empty template parameter lists. If so, a partial class template specialization is defined.

A completely specialized class has the following characteristics:

- The class template specialization must be provided after the generic class template definition. As it is a specialization the compiler must first have seen the original class template;
- The completely specialized class template's template parameter list is empty;
• All of the class’s template parameters are given explicit type names or (for the non-type parameters) explicit values. These explicitations are provided in a template parameter specification list (surrounded by angle brackets) that is inserted immediately after the specialized template’s class name;

• All members of the specialized class template use specialized types and values where original template parameters are used in the original template definition;

• All original template’s members (maybe with the exception of some constructors) should be redefined by the specialization. If a member is left out of the specialization, it cannot be used for a specialized class template object;

• The specialization may define additional members (but maybe shouldn’t as it breaks the one-to-one correspondence between the original and specialized class template);

• Member functions of specialized class templates may be declared by the specializing class and implemented below their class interface. If their implementations follow the class interface they may not begin with a template <> header, but must immediately start with the member function's header.

22.3.1 Example of a class specialization

Here is an example of a completely specialized CirQue class, specialized for a vector<int>. All members of the specialized class are declared, but only non-trivial implementations of its members are provided. The specialized class uses a copy of the vector passed to the constructor and implements a circular queue using its vector data member:

```cpp
#ifndef INCLUDED_CIRQUEVECTOR_H_
#define INCLUDED_CIRQUEVECTOR_H_

#include <vector>
#include "cirque.h"

template<>
class CirQue<std::vector<int>>
{
  typedef std::vector<int> IntVect;
  IntVect d_data;
  size_t d_size;

  typedef IntVect::iterator iterator;
  iterator d_front;
  iterator d_back;

public:
  typedef int value_type;
  typedef value_type const &const_reference;

  enum Exception
  {
    EMPTY,
    FULL
  };
```
CirQue();
CirQue(IntVect const &iv);
CirQue(CirQue<IntVect> const &other);

CirQue &operator=(CirQue<IntVect> const &other);

int &back();
int &front();
bool empty() const;
bool full() const;
size_t maxSize() const;
size_t size() const;
void pop_front();
void push_back(int const &object);
void swap(CirQue<IntVect> &other);

private:
    iterator inc(iterator const &iter);
};

CirQue<std::vector<int>>::CirQue()
:    d_size(0)
{}
CirQue<std::vector<int>>::CirQue(IntVect const &iv)
:    d_data(iv),
     d_size(iv.size()),
     d_front(d_data.begin()),
     d_back(d_data.begin())
{}
CirQue<std::vector<int>>::CirQue(CirQue<IntVect> const &other)
:    d_data(other.d_data),
     d_size(other.d_size),
     d_front(d_data.begin() + (other.d_front - other.d_data.begin())),
     d_back(d_data.begin() + (other.d_back - other.d_data.begin()))
{}
CirQue<std::vector<int>> &CirQue<std::vector<int>>::operator=(
    CirQue<IntVect> const &rhs)
{
    CirQue<IntVect> tmp(rhs);
    swap(tmp);
}
void CirQue<std::vector<int>>::swap(CirQue<IntVect> &other)
{
    char tmp[sizeof(CirQue<IntVect>)];
    memcpy(tmp, &other, sizeof(CirQue<IntVect>));
    memcpy(&other, this, sizeof(CirQue<IntVect>));
    memcpy(this, tmp, sizeof(CirQue<IntVect>));
}
void CirQue<std::vector<int>>::pop_front()
if (d_size == 0)
    throw EMPTY;

d_front = inc(d_front);
--d_size;
}

void CirQue< std::vector<int> >::push_back(int const &object)
{
    if (d_size == d_data.size())
        throw FULL;

    *d_back = object;
    d_back = inc(d_back);
    ++d_size;
}

inline int &CirQue< std::vector<int> >::back()
{
    return d_back == d_data.begin() ? d_data.back() : d_back[-1];
}

inline int &CirQue< std::vector<int> >::front()
{
    return *d_front;
}

CirQue< std::vector<int> >::iterator CirQue< std::vector<int> >::inc(
    CirQue< std::vector<int> >::iterator const &iter)
{
    iterator tmp(iter + 1);
    tmp = tmp == d_data.end() ? d_data.begin() : tmp;
    return tmp;
}

#endif

The next example shows how to use the specialized CirQue class:

static int iv[] = {1, 2, 3, 4, 5};

int main()
{
    vector<int> vi(iv, iv + 5);
    CirQue<vector<int>> ci(vi);

    cout << ci.size() << ' ' << ci.front() << ' ' << ci.back() << '
';
    ci.pop_front();
    ci.pop_front();

    CirQue<vector<int>> cp;
    cp = ci;
    cout << cp.size() << ' ' << cp.front() << ' ' << cp.back() << '
';
    cp.push_back(6);
    cout << cp.size() << ' ' << cp.front() << ' ' << cp.back() << '
';
}
PARTIAL SPECIALIZATIONS

In the previous section class template specializations were introduced. In this section we’ll introduce a variant of this specialization, both in number and type of template parameters that are specialized. *

Partial specializations may be defined for class templates having multiple template parameters. Function templates cannot be partially specialized.

With partial specializations a subset (any subset) of template type parameters are given specific values. It is also possible to use a class template partial specialization when the intent is to specialize the class template, but to parameterize the data type that is processed by the specialization.

To start our discussion with an example of the latter use of a partial class template specialization consider the class CirQue<vector<int>> developed in the previous section. When designing CirQue<vector<int>> you may have asked yourself how many specializations you’d have to implement. One for vector<int>, one for vector<string>, one for vector<double>? As long as the data types handled by the vector used by the class CirQue<vector<...>> behaves like an int (i.e., is a value-type of class) the answer is: zero. Instead of defining full specializations for each new data type the data type itself can be parameterized, resulting in a partial specialization:

```
template <typename Data>
class CirQue<std::vector<Data>>
{
    ...
};
```

The above class is a specialization as a template argument list is appended to the CirQue class name. But as the class template itself has a non-empty template parameter list it is in fact recognized as a partial specialization. There is one characteristic that distinguishes the implementation (subsequent to the class template’s interface) of a class template member function of a partial specialization from the implementation of a member function of a full specialization. Implementations of partially specialized class template member functions receive a template header. No template headers are used when implementing fully specialized class template members.

Implementing the partial specialization for CirQue is not difficult and is left as an exercise for the reader (hints: simply change int into Data in the CirQue<vector<int>> specialization of the previous section). Remember to prefix the type iterator by typename (as in typedef typename DataVect::iterator iterator) (as discussed in section 22.2.1).

In the next subsections we’ll concentrate on specializing class template non-type template parameters. These partial specializations are now illustrated using some simple concepts defined in matrix algebra, a branch of linear algebra.
22.4.1 Intermezzo: some simple matrix algebraic concepts

In this section some simple matrix algebraic terms are introduced. These terms are used in the next sections to illustrate and discuss partial specializations of class templates. Readers proficient in matrix algebra may skip this section without loss of continuity.

A matrix is commonly thought of as a table of some rows and columns, filled with numbers. Immediately we recognize an opening for using templates: the numbers might be plain double values, but they could also be complex numbers, for which our complex container (cf. section 12.5) might prove useful. Our class template is therefore provided with a DataType template type parameter. It is specified when a matrix is constructed. Some simple matrices using double values, are:

\[
\begin{array}{ccc}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{array}
\] An identity matrix, (a 3 x 3 matrix).

\[
\begin{array}{cccc}
1.2 & 0 & 0 & 0 \\
0.5 & 3.5 & 18 & 23 \\
\end{array}
\] A rectangular matrix, (a 2 x 4 matrix).

\[
\begin{array}{cccc}
1 & 2 & 4 & 8 \\
\end{array}
\] A matrix of one row, (a 1 x 4 matrix), also known as a ‘row vector’ of 4 elements.

Various operations are defined on matrices. They may, for example be added, subtracted or multiplied. Here we will not focus on these operations. Rather, we concentrate on some simple operations: computing marginals and sums.

Marginals are the sums of row elements or the sums of column elements of a matrix. These two kinds of marginals are also known as, respectively, row marginals and column marginals.

- **Row marginals** are obtained by adding, for each row, all the row’s elements and putting these (Rows) sums in corresponding elements of a (column) vector of Rows elements.

- **Column marginals** are obtained by adding, for each column, all the column’s elements and putting these (Columns) sums in corresponding elements of a (row) vector of Columns elements.

- The sum of all elements of a matrix can of course be computed as the sum of the elements of one of its marginals.

The following example shows a matrix, its marginals, and the sum of its values:

<table>
<thead>
<tr>
<th>matrix</th>
<th>row marginals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
<td>6</td>
</tr>
<tr>
<td>4 5 6</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>column marginals</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 7 9</td>
</tr>
<tr>
<td>21 (sum)</td>
</tr>
</tbody>
</table>
22.4.2 The Matrix class template

We'll start out by introducing a class template defining a matrix. Having defined this class template we'll continue with defining several specializations.

Since matrices consist of well defined numbers of rows and columns (the dimensions of the matrix), that normally do not change when matrices are used, we might consider specifying their values as template non-type parameters. The \texttt{DataType = double} will be used in the majority of cases. Therefore, \texttt{double} can be selected as the template's default type argument. Since it's a sensible default, the \texttt{DataType} template type parameter is used last in the template type parameter list.

Our template class \texttt{Matrix} begins its life as:

\begin{verbatim}
template <size_t Rows, size_t Columns, typename DataType = double>
class Matrix
...
\end{verbatim}

What do we want our class template to offer?

- It needs a place to store its matrix elements. This can be defined as an array of \texttt{Rows}' rows each containing \texttt{Columns}' elements of type \texttt{DataType}. It can be an array, rather than a pointer, since the matrix' dimensions are known \textit{a priori}. Since a vector of \texttt{Columns} elements (a row of the matrix), as well as a vector of \texttt{Row} elements (a column of the matrix) is often used, the class could use \texttt{typedef}s to represent them. The class interface's initial section thus contains:

\begin{verbatim}
typedef Matrix<1, Columns, DataType> MatrixRow;
typedef Matrix<Rows, 1, DataType> MatrixColumn;
MatrixRow d_matrix[Rows];
\end{verbatim}

- It should offer constructors: a default constructor and (e.g.,) a constructor initializing the matrix from a stream. A copy or move constructor is not required as the class does not use pointers. Likewise, no overloaded assignment operator or destructor is required. Implementations:

\begin{verbatim}
template <size_t Rows, size_t Columns, typename DataType>
Matrix<Rows, Columns, DataType>::Matrix() 
{ 
    std::fill(d_matrix, d_matrix + Rows, MatrixRow());
}
template <size_t Rows, size_t Columns, typename DataType>
Matrix<Rows, Columns, DataType>::Matrix(std::istream &str) 
{ 
    for (size_t row = 0; row < Rows; row++)
        for (size_t col = 0; col < Columns; col++)
            str >> d_matrix[row][col];
}
\end{verbatim}

- The class's \texttt{operator[]} member (and its \texttt{const} variant) only handles the first index, returning a reference to a complete \texttt{MatrixRow}. How elements in a \texttt{MatrixRow} can be retrieved is shortly covered. To keep the example simple, no array bound check has been implemented:

\begin{verbatim}
template <size_t Rows, size_t Columns, typename DataType>
Matrix<1, Columns, DataType>
&Matrix<Rows, Columns, DataType>::operator[](size_t idx)
\end{verbatim}
Now we get to the interesting parts: computing marginals and the sum of all elements in a Matrix. We'll define the type `MatrixColumn` as the type containing the row marginals of a matrix, and the type `MatrixRow` as the type containing the column marginals of a matrix. There is also the sum of all the elements of a matrix. This sum of all the elements of a matrix is a number that itself can be thought of as a 1 x 1 matrix.

Marginals can be considered as special forms of matrices. To represent these marginals we can construct partial specializations defining the class templates `MatrixRow` and `MatrixColumn` objects; and we construct a partial specialization handling 1 x 1 matrices. These partial specializations are used to compute marginals and the sum of all the elements of a matrix. Before concentrating on these partial specializations themselves we'll use them here to implement the members computing the marginals and the sum of all elements of a matrix:

```cpp
template <size_t Rows, size_t Columns, typename DataType>
Matrix<1, Columns, DataType> Matrix<Rows, Columns, DataType>::columnMarginals() const
{
    return MatrixRow(*this);
}

template <size_t Rows, size_t Columns, typename DataType>
Matrix<1, 1, DataType> Matrix<Rows, Columns, DataType>::rowMarginals() const
{
    return MatrixColumn(*this);
}

template <size_t Rows, size_t Columns, typename DataType>
DataType Matrix<Rows, Columns, DataType>::sum() const
{
    return rowMarginals().sum();
}
```

### 22.4.3 The MatrixRow partial specialization

Class template partial specializations can be defined for any (subset) of template parameters. They can be defined for template type parameters and for template non-type parameters alike. Our first partial specialization defines a row of a generic Matrix, mainly (but not only) used for the construction of column marginals. Here is how such a partial specialization is designed:

- The partial specialization starts with a template header defining all template parameters that are not specialized in the partial specialization. This template header cannot specify any defaults (like `DataType = double`) since defaults were already specified by the generic class template definition. The specialization must follow the definition of the generic class template's definition, or the compiler complains that it doesn't know what class is being specialized. Following the template header, the class's interface starts. It's a class template (partial) specialization so the class name must be followed by a template argument list specifying the template arguments used by the partial specialization. The arguments specify explicit types or values for some of the template's parameters. Remaining types are simply copied from the
class template partial specialization's template parameter list. E.g., the MatrixRow specialization specifies 1 for the generic class template's Rows non-type parameter (as we're talking here about a single row). Both Columns and DataType remain to be specified. The MatrixRow partial specialization therefore starts as follows:

```
template <size_t Columns, typename DataType> // no default allowed
class Matrix<1, Columns, DataType>
```

- A MatrixRow holds the data of a single row. So it needs a data member storing Columns values of type DataType. Since Columns is a constant value, the d_row data member can be defined as an array:

```
DataType d_column[Columns];
```

- The class template partial specialization's constructors require some attention. The default constructor is simple. It merely initializes the MatrixRow's data elements using DataType's default constructor:

```
template <size_t Columns, typename DataType>
Matrix<1, Columns, DataType>::Matrix()
{
    std::fill(d_column, d_column + Columns, DataType());
}
```

Another constructor is needed initializing a MatrixRow object with the column marginals of a generic Matrix object. This requires us to provide the constructor with a non-specialized Matrix parameter.

The rule of thumb here is to define a member template that allows us to keep the general nature of the parameter. The generic Matrix template requires three template parameters. Two of these were already provided by the template specialization. The third parameter is mentioned in the member template's template header. Since this parameter refers to the number of rows of the generic matrix it is simply called Rows.

Here then is the implementation of the second constructor, initializing the MatrixRow's data with the column marginals of a generic Matrix object:

```
template <size_t Columns, typename DataType>
template <size_t Rows>
Matrix<1, Columns, DataType>::Matrix(
    Matrix<Rows, Columns, DataType> const & matrix)
{
    std::fill(d_column, d_column + Columns, DataType());

    for (size_t col = 0; col < Columns; col++)
        for (size_t row = 0; row < Rows; row++)
            d_column[col] += matrix[row][col];
}
```

The constructor's parameter is a reference to a Matrix template using the additional Row template parameter as well as the template parameters of the partial specialization.

- We don't really require additional members to satisfy our current needs. To access the data elements of the MatrixRow an overloaded operator[]() is of course useful. Again, the const variant can be implemented like the non-const variant. Here is its implementation:

```
template <size_t Columns, typename DataType>
```
Now that we have defined the generic `Matrix` class and the partial specialization defining a single row the compiler selects the row’s specialization whenever a `Matrix` is defined using `Row = 1`. For example:

```
Matrix<4, 6> matrix; // generic Matrix template is used
Matrix<1, 6> row;   // partial specialization is used
```

### 22.4.4 The `MatrixColumn` partial specialization

The partial specialization for a `MatrixColumn` is constructed similarly. Let’s present its highlights (the full `Matrix` class template definition as well as all its specializations are provided in the `cplusplus.yo.zip` archive (at SourceForge) in the file `yo/classtemplates/examples/matrix.h`):

- The class template partial specialization once again starts with a template header. Now the class interface specifies a fixed value for the second template parameter of the generic class template. This illustrates that we can construct partial specializations for every single template parameter; not just for the first or the last:

```
template <size_t Rows, typename DataType>
class Matrix<Rows, 1, DataType>
```

- Its constructors are implemented completely analogously to the way the `MatrixRow` constructors were implemented. Their implementations are left as an exercise to the reader (and they can be found in `matrix.h`).

- An additional member `sum` is defined to compute the sum of the elements of a `MatrixColumn` vector. It’s simply implemented using the `accumulate` generic algorithm:

```
template <size_t Rows, typename DataType>
DataType Matrix<Rows, 1, DataType>::sum()
{
    return std::accumulate(d_row, d_row + Rows, DataType());
}
```

### 22.4.5 The 1x1 matrix: avoid ambiguity

The reader might wonder what happens if we define the following matrix:

```
Matrix<1, 1> cell;
```

Is this a `MatrixRow` or a `MatrixColumn` specialization? The answer is: neither.

It’s ambiguous, precisely because both the columns and the rows could be used with a (different) template partial specialization. If such a Matrix is actually required, yet another specialized template must be designed.

Since this template specialization can be useful to obtain the sum of the elements of a Matrix, it’s covered here as well.

- This class template partial specialization also needs a template header, this time only specifying \texttt{DataType}. The class definition specifies two fixed values: 1 for the number of rows and 1 for the number of columns:

  \begin{verbatim}
  template <typename \texttt{DataType}>
  class Matrix<1, 1, \texttt{DataType}>
  \end{verbatim}

- The specialization defines the usual batch of constructors. Constructors expecting a more generic Matrix type are again implemented as member templates. For example:

  \begin{verbatim}
  template <typename \texttt{DataType}>
  template <\texttt{size_t} \texttt{Rows}, \texttt{size_t} \texttt{Columns}>
  Matrix<1, 1, \texttt{DataType}>::Matrix(
      Matrix<\texttt{Rows}, \texttt{Columns}, \texttt{DataType}> const &\texttt{matrix})
  :
    d_cell(\texttt{matrix.rowMarginals()}.sum())
  {}

  template <typename \texttt{DataType}>
  template <\texttt{size_t} \texttt{Rows}>
  Matrix<1, 1, \texttt{DataType}>::Matrix(Matrix<\texttt{Rows}, 1, \texttt{DataType}> const &\texttt{matrix})
  :
    d_cell(\texttt{matrix.sum()})
  {}
  \end{verbatim}

- Since Matrix<1, 1> is basically a wrapper around a \texttt{DataType} value, we need members to access that latter value. A type conversion operator might be useful, but we also defined a \texttt{get} member to obtain the value if the conversion operator isn’t used by the compiler (which happens when the compiler is given a choice, see section 11.3). Here are the accessors (leaving out their const variants):

  \begin{verbatim}
  template <typename \texttt{DataType}>
  Matrix<1, 1, \texttt{DataType}>::\texttt{operator} \texttt{DataType} &() {
    return d_cell;
  }

  template <typename \texttt{DataType}>
  \texttt{DataType} &Matrix<1, 1, \texttt{DataType}>::\texttt{get}() {
    return d_cell;
  }
  \end{verbatim}

Finally, the main function shown below illustrates how the Matrix class template and its partial specializations can be used:

\begin{verbatim}
#include <iostream>

Finally, the main function shown below illustrates how the Matrix class template and its partial specializations can be used:

```cpp
#include <iostream>

```
```cpp
#include "matrix.h"
using namespace std;

int main(int argc, char **argv)
{
    Matrix<3, 2> matrix(cin);

    Matrix<1, 2> colMargins(matrix);
    cout << "Column marginals:\n";
    cout << colMargins[0] << " " << colMargins[1] << '\n';

    Matrix<3, 1> rowMargins(matrix);
    cout << "Row marginals:\n";
    for (size_t idx = 0; idx < 3; idx++)
        cout << rowMargins[idx] << '\n';

    cout << "Sum total: " << Matrix<1, 1>(matrix) << '\n';
    return 0;
}
/*
Generated output from input: 1 2 3 4 5 6

Column marginals:
9 12
Row marginals:
3
7
11
Sum total: 21
*/

22.5 Variadic templates

Up to this point we've only encountered templates having a fixed number of template parameters. The C++11 standard extends this with variadic templates.

Variadic templates are defined for function templates and for class templates. Variadic templates allow us to specify an arbitrary number of template arguments of any type.

Variadic templates were added to the language to prevent us from having to define many overloaded templates and to be able to create type safe variadic functions.

Although C (and C++) support variadic functions, their use has always been deprecated in C++ as those functions are notoriously type-unsafe. Variadic function templates can be used to process objects that until now couldn't be processed properly by C-style variadic functions.

Template headers of variadic templates use the phrase typename ...Params (Params being a formal name). A variadic class template Variadic could be declared as follows:

```cpp
template<typename ...Params> class Variadic;
```

Assuming the class template's definition is available then this template can be instantiated using any number of template arguments. Example:
class Variadic<
    int,
    std::vector<int>,
    std::map<std::string, std::vector<int>>
> v1;

The template argument list of a variadic template can also be empty. Example:

class Variadic<> empty;

If this is considered undesirable using an empty template argument list can be prevented by providing one or more fixed parameters. Example:

template<typename First, typename ...Rest>
class tuple;

C’s function printf is a well-known example of a type-unsafe function. It is turned into a typesafe function when it is implemented as a variadic function template. Not only does this turn the function into a type-safe function but it is also automatically extended to accept any type that can be defined by C++. Here is a possible declaration of a variadic function template printcpp:

template<typename ...Params>
void printcpp(std::string const &strFormat, Params ...parameters);

The ellipsis (...) used in the declaration serves two purposes:

- In the template header it is written to the left of a template parameter name where it declares a parameter pack. A parameter pack allows us to specify any number of template arguments when instantiating the template. Parameter packs can be used to bind type and non-type template arguments to template parameters.

- In a template implementation it appears to the right of the template pack’s parameter name. In that case it represents a series of template arguments that are subsequently matched with a function parameter that in turn is provided to the right of the ellipsis. Here the ellipsis is known as the unpack operator as it ‘unpacks’ a series of arguments in a function’s argument list thereby implicitly defining its parameters.

C++ offers no syntax to access the individual template arguments directly. However, the arguments can be visited recursively. An example is provided in the next section. The number of arguments is determined using a new invocation of the sizeof operator:

template<typename ...Params>
struct StructName
{
    enum: size_t { s_size = sizeof ...(Params) };
};

// StructName<int, char>::s_size -- initialized to 2

### 22.5.1 Defining and using variadic templates

The arguments associated with a variadic template parameter are not directly available to the implementation of a function or class template. We have to resort to other means to obtain them.
By defining a partial specialization of a variadic template, explicitly defining an additional template type parameter, we can associate the first template argument of a parameter pack with this additional (first) type parameter. The setup of such a variadic function template (e.g., `printcpp`, see the previous section) is as follows:

- The `printcpp` function receives at least a format string. Following the format string any number of additional arguments may be specified.
- If there are no arguments trailing the format string then there is no need to use a function template. An overloaded (non-template) function is defined to handle this situation.
- A variadic function template handles all remaining situations. In this case there is always at least one argument trailing the format string. That argument's type is matched with the variadic template function's first (ordinary) template type parameter `First`. The types of any remaining arguments are bound to the template function's second template parameter, which is a parameter pack.
- The variadic function template processes the argument trailing the format string. Then it recursively calls itself passing the format string and the parameter pack to the recursive call.
- If the recursive call merely receives the format string the overloaded (non-template) function is called (cf. section 21.14) ending the recursion. Otherwise the parameter pack's first argument is matched with the recursive call's `First` parameter. As this reduces the size of the recursive call's parameter pack the recursion eventually stops.

The overloaded non-template function prints the remainder of the format string, *en passant* checking for any left-over format specifications:

```cpp
void printcpp(string const &format)
{
    size_t left = 0;
    size_t right = 0;

    while (true)
    {
        if ((right = format.find('%', right)) == string::npos)
            break;
        if (format.find("%\", right) != right)
            throw std::runtime_error("printcpp: missing arguments");
        ++right;
        cout << format.substr(left, right - left);
        left = ++right;
    }
    cout << format.substr(left);
}
```

Here is the variadic function template's implementation:

```cpp
template<typename First, typename ...Params>
void printcpp(std::string const &format, First value, Params ...params)
{
    size_t left = 0;
    size_t right = 0;
    ```
while (true)
{
    if ((right = format.find('%', right)) == string::npos) // 1
        throw std::runtime_error("printcpp: too many arguments");

    if (format.find("%%", right) != right) // 2
        break;

    ++right;
    cout << format.substr(left, right - left);
    left = ++right;
}
cout << format.substr(left, right - left) << value;
printcpp(format.substr(right + 1), params...);

• At 1 the format string is searched for a parameter specification %. If none is found then the
  function is called with too many arguments and it throws an exception;

• At 2 it verifies that it has not encountered %%. If only a single % has been seen the while-loop
  ends, the format string is inserted into cout up to the % followed by value, and the recursive
  call receives the remaining part of the format string as well as the remaining parameter pack;

• If % was seen the format string is inserted up to the second %, which is ignored, and processing
  of the format string continues beyond the %.

Make sure that the overloaded function is at least declared before the compiler processes the function
template's definition or it won't call the overloaded function printcpp when compiling the function
template.

Different from C’s printf function printcpp only recognizes % and %%% as format specifiers. The
above implementation does not recognize, e.g., field widths. Type specifiers like %c and %x are of
course not needed as ostream's insertion operator is aware of the types of the arguments that are
inserted into the ostream. Extending the format specifiers so that field widths etc. are recognized
by this printcpp implementation is left as an exercise to the reader. Here is an example showing
how printcpp can be called:

    printcpp("Hello % with %main%% called with % args
               " and a string showing %\n",
            "world", argc, string("A String"));

22.5.2 Perfect forwarding

Consider string’s member insert. String::insert has several overloaded implementations. It
can be used to insert text (completely or partially) provided by a string or by a char const *
argument; to insert single characters a specified number of times; iterators can be used to specify
the range of characters to be inserted; etc., etc.. All in all, string offers as many as five overloaded
insert members.

Assume the existence of a class Inserter that is used to insert information into all kinds of objects.
Such a class could have a string data member into which information can be inserted. Inserter's
interface only partially has to copy string's interface to realize this: only string::insert's inter-
faces must be duplicated. The members duplicating interfaces often contain one statement (calling
the appropriate member function of the object's data member) and are for this reason often implemented in-line. Such wrapper functions merely forward their parameters to the matching member functions of the object's data member.

Another example is found in factory functions that also frequently forward their parameters to the constructors of objects that they return.

Before the C++11 standard the interfaces of overloaded functions needed to be duplicated by the forwarding entity: Inserter needed to duplicate the interfaces of all five string::insert members; a factory function needed to duplicate the interfaces of the constructors of the class of the objects it returned.

The C++11 standard simplifies and generalizes forwarding of parameters by offering perfect forwarding, implemented through rvalue references and variadic templates. With perfect forwarding the arguments passed to functions are 'perfectly forwarded' to nested functions. Forwarding is called perfect as the arguments are forwarded in a type-safe way. To use perfect forwarding nested functions must define parameter lists matching the forwarding parameters both in types and number.

Perfect forwarding is easily implemented:

- The forwarding function is defined as a template (usually a variadic template, but single argument forwarding is also possible. To define and forward a single argument omit the ellipsis from the following code);
- The forwarding function's parameter list is an rvalue reference parameter pack (e.g., Params &&...params);
- std::forward is used to forward the forwarding function's arguments to the nested function, keeping track of their types and number. Before forward can be used the <utility> header file must be included.
- The nested function is called using this stanza to specify its arguments: std::forward<Params>(params) ....

In the next example perfect forwarding is used to implement one member Inserter::insert that can be used to call any of the five overloaded string::insert members. The insert function that's actually called now simply depends on the types and number of arguments that are passed to Inserter::insert:

```cpp
class Inserter
{
    std::string d_str; // somehow initialized

    public:
        // constructors not implemented,
        // but see below
        Inserter();
        Inserter(std::string const &str);
        Inserter(Inserter const &other);
        Inserter(Inserter &&other);

        template<typename ...Params>
        void insert(Params &&...params)
        {
            d_str.insert(std::forward<Params>(params)...);
        }
};
```
A factory function returning an \texttt{Inserter} can also easily be implemented using perfect forwarding. Rather than defining four overloaded factory functions a single one now suffices. By providing the factory function with an additional template type parameter specifying the class of the object to construct the factory function is turned into a completely general factory function:

\begin{verbatim}
template <typename Class, typename ...Params>
Class factory(Params &&...params)
{
    return Class(std::forward<Params>(params)...);
}
\end{verbatim}

Here are some examples showing its use:

\begin{verbatim}
Inserter inserter(factory<Inserter>("hello"));
string delimiter(factory<string>(10, '='));
Inserter copy(factory<Inserter>(inserter));
\end{verbatim}

The function \texttt{std::forward} is provided by the standard library. It performs no magic, but merely returns \texttt{params} as a nameless object. That way it acts like \texttt{std::move} that also removes the name from an object, returning it as a nameless object. The unpack operator has nothing to do with the use of \texttt{forward} but merely tells the compiler to apply \texttt{forward} to each of the arguments in turn. Thus it behaves similarly to C's ellipsis operator used by variadic functions.

Perfect forwarding was introduced in section 21.4.5: a template function defining a \texttt{Type \&\&param}, with \texttt{Type} being a template type parameter converts \texttt{Type \&\&} to \texttt{Tp \&} if the function is called with an argument of type \texttt{Tp \&}. Otherwise it binds \texttt{Type} to \texttt{Tp}, with \texttt{param} being defined as \texttt{Tp \&\&param}. As a result an \texttt{lvalue} argument binds to an lvalue-type (\texttt{Tp \&}), while an \texttt{rvalue} argument binds to an rvalue-type (\texttt{Tp \&\&}).

The function \texttt{std::forward} merely passes the argument (and its type) on to the called function or object. Here is its simplified implementation:

\begin{verbatim}
typedef <type T>
T \&\&forward(T \&\&a)
{
    return a;
}
\end{verbatim}

Since \texttt{T \&\&} turns into an lvalue reference when \texttt{forward} is called with an lvalue (or lvalue reference) and remains an rvalue reference if \texttt{forward} is called with an rvalue reference, and since \texttt{forward} (like \texttt{std::move}) anonymizes the variable passed as argument to \texttt{forward}, the argument value is forwarded while passing its type from the function's parameter to the called function's argument.

This is called \textit{perfect forwarding} as the nested function can only be called if the types of the arguments that were used when calling the 'outer' function (e.g., \texttt{factory}) exactly match the types of the parameters of the nested function (e.g., \texttt{Class}'s constructor). Perfect forwarding therefore is a tool to realize type safety.

A cosmetic improvement to \texttt{forward} requires users of \texttt{forward} to specify the type to use rather than to have the compiler deduce the type as a result of the function template parameter type deduction's process. This is realized by a small support struct template:

\begin{verbatim}
template <typename T>
\end{verbatim}
struct identity
{
    typedef T type;
};

This struct merely defines identity::type as T, but as it is a struct it must be specified explicitly. It cannot be determined from the function's argument itself. The subtle modification to the above implementation of forward thus becomes (cf. section 22.2.1 for an explanation of the use of typename):

typedef <type T>
T &&forward(typename identity<T>::type &&arg)
{
    return arg;
}

Now forward must explicitly state arg's type, as in:

std::forward<Params>(params)

Using the std::forward function and the rvalue reference specification is not restricted to the context of parameter packs. Because of the special way rvalue references to template type parameters are treated (cf. section 21.4.5) they can profitably be used to forward individual function parameters as well. Here is an example showing how an argument to a function can be forwarded from a template to a function that is itself passed to the template as a pointer to an (unspecified) function:

template<typename Fun, typename ArgType>
void caller(Fun fun, ArgType &&arg)
{
    fun(std::forward<ArgType>(arg));
}

A function display(ostream &out) and increment(int &x) may now both be called through caller. Example:

caller(display, cout);
int x = 0;
caller(increment, x);

22.5.2.1 References to references

Whenever a class template's parameter is a reference the compiler can't distinguish between member functions overloading const and non-const references. Here is an example of a program that won't compile:

template <typename T>
class Wrap
{
    public:
        Wrap(T const &tr);
22.5. VARIADIC TEMPLATES

```cpp
Wrap(T &tr);
}
```

```cpp
int main()
{
    int i;
    Wrap<int &> wrap(i);
}
```

and the compiler generates an error message like

`'Wrap<T>::Wrap(T&) [with T = int&]' cannot be overloaded with 'Wrap<T>::Wrap(const T&) [with T = int&]'`

Earlier, in section 19.2.2 a program was shown using a support function having signature

```cpp
int stringcasecmp(string lhs, string rhs)
```

In that section it was noted that this function did introduce inefficiency, but that an improved signature, using `string const &` parameters fails to compile. With a function declaring a `string const &` parameter `std::bind2nd` (cf. section 18.1.4) generates an error like the one mentioned here. Following the call

```cpp
auto pos = find_if(v1.begin(), v1.end(),
                   not1(bind2nd(ptr_fun(stringcasecmp), target))
             );
```

that was used in section 19.2.2 as well as the definition of `bind2nd` (usually found in the system header file binders.h) we encounter the following series of instantiated templates:

- First, `ptr_fun` was instantiated, receiving `stringcasecmp`. The function template `ptr_fun` returns a `pointer_to_binary_function`, for which its `first_argument_type` and `second_argument_type` are determined as `string const &` if `stringcasecmp` defines `string const &` parameters;
- Next, the `pointer_to_binary_function` object is passed to `bind2nd`. As you may have guessed, we’re already on dangerous grounds as `pointer_to_binary_function` is a class template defining reference template type parameters;
- The `bind2nd` function template merely creates and returns a `binder2nd` object, initializing it with the `pointer_to_binary_function` object and the type of the second parameter of `stringcasecmp` (which can be ignored in the current discussion);
- The `binder2nd` class template object is a functor defining two overloaded function operators (using declarations and simplified type names to ease focusing on the essence):
  ```cpp
  result_type operator()(first_argument_type const &x) const;
  result_type operator()(first_argument_type &x) const;
  ```

As `first_argument_type` is a template type parameter deducted to be `string const &` we have overloaded functions using reference parameters and a template type parameter which itself is a reference parameter. As a consequence the ‘cannot be overloaded’ error message results and `stringcasecmp` cannot be defined as a function having `string const &` parameters.
Before the arrival of the C++11 standard this problem could not elegantly be solved, as pre-C++11 versions of C++ did not offer perfect forwarding. The solution requires a reformulation of `bind2nd` and `binder2nd`, exploiting features offered by perfect forwarding. Other adaptors defined by the STL may require comparable reformulations. The modifications required for `bind2nd` and `binder2nd` are covered here; it's up to the reader to implement such modifications for other adaptors when appropriate.

std::bind2nd is rewritten here as `Bind2nd`. It's actually a copy of `bind2nd`, but this time creating `Binder2nd` rather than `binder2nd`. As the (non-configurable) venom is in `bind2nd`'s `binder2nd` call an alternative for `bind2nd` is needed to avoid calling `binder2nd`. Alternative approaches are also possible. A class template specialization for `binder2nd` could be developed expecting a particular class which is a wrapper around `pointer_to_binary_function`. It wouldn't make much of a difference, as in that case `ptr_fun` would have to be rewritten as well.

So we stick to `Bind2nd` having the following obvious implementation (all functions defined in-class to save some space):

```cpp
template<typename Operation, typename SecondArg>
inline Binder2nd<Operation> Bind2nd(Operation const &operation,
    SecondArg const &arg2)
{
    return Binder2nd<Operation>(
            operation,
            typename Operation::second_argument_type(arg2)
    );
}
```

The meat of the solution is of course in `Binder2nd`, now using perfect forwarding. There is now only one function operator member (operator()) which is now defined as a function template, using the perfect forwarding incantation. Its body forwards the arguments that were actually passed to the function call operator as well as `Binder2nd`'s (fixed) second argument to the 'operation', which is `stringcasecmp`. Please note that the forwarded arguments can very well be followed by additional arguments (here: `d_arg2`). In `Binder2nd`'s implementation there's no guarantee that there will actually only be one perfectly forwarded argument. Although such a guarantee could easily be built-in (cf. section 23.9) it really isn't required as the compiler detects whether there are indeed two arguments being passed to `stringcasecmp`, generating an error message if not. Here is `Binder2nd`'s implementation:

```cpp
template<typename Operation>
class Binder2nd:
    public std::unary_function<typename Operation::first_argument_type,
        typename Operation::result_type>
{
    typedef typename Operation::second_argument_type SecondArg;

    protected:
        Operation d_operation;
        SecondArg d_arg2;

    public:
        Binder2nd(Operation const &operation, SecondArg const &arg2)
            :
                d_operation(operation),
                d_arg2(arg2)
        
```
The program shown in section 19.2.2 remains unaltered but for the use of `bind2nd`, which now becomes `Bind2nd`:

```cpp
... auto pos = find_if(
    v1.begin(), v1.end(),
    not1( Bind2nd(ptr_fun(stringicmp), target) )
);
```

The two programs produce identical output, but the program developed here is a lot more efficient than the one developed originally, in section 19.2.2.

### 22.5.3 The unpack operator

The **unpack operator** is used to obtain template arguments in many situations. No mechanism other than recursion (as shown in section 22.5.1) is available to obtain the individual types and values of a variadic template.

The unpack operator can also be used to define template classes that are derived from any number of base classes. Here is how it’s done:

```cpp
template <typename ...BaseClasses>
class Combi: public BaseClasses...
{
    public:
        // specify base class objects
        // to its constructor using
        // perfect forwarding
    Combi(BaseClasses &&...baseClasses)
    :
        BaseClasses(baseClasses)... // use base class initializers
    } // for each of the base
}; // classes
```

This allows us to define classes that combine the features of any number of other classes. If the class `Combi` is derived of classes `A`, `B`, and `C` then `Combi` is an `A`, `B`, and `C`. It should of course be given a virtual destructor. A `Combi` object can be passed to functions expecting pointers or references to any of its base class type objects. Here is an example defining `Combi` as a class derived from a vector of complex numbers, a string and a pair of ints and doubles (using uniform initializers in a sequence matching the sequence of the types specified for the used `Combi` type):

```cpp
typedef Combi<
```
The same construction can also be used to define template data members supporting variadic type lists such as tuples (cf. section 22.6). Such a class could be designed along these lines:

```cpp
template <typename ...Types>
struct Multi
{
    std::tuple<Types...> d_tup; // define tuple for Types types
    Multi(Types ...types) // initialize d_tup from Multi’s arguments
    : d_tup(std::forward<Types>(types)...) {}
};
```

The ellipses that are used when forwarding parameter packs are essential. The compiler considers their omission an error. In the following `struct` definition it was the intent of the programmer to pass a parameter pack on to a nested object construction but ellipses were omitted while specifying the template parameters, resulting in a `parameter packs not expanded with ‘..’` error message:

```cpp
template <int size, typename ...List>
struct Call
{
    Call(List &&...list) // call with ellipses omitted
    {
        Call<size - 1, List &&> call(std::forward<List>(list)...);
    }
};
```

Instead of the above definition of the `call` object the programmer should have used:

```cpp
Call<size - 1, List &&...> call(std::forward<List>(list)...);
```

### 22.5.4 Non-type variadic templates

Variadic templates not necessarily define template types. Non-types can also be used with variadic templates. The following function template accepts any series of `int` values, forwarding these values to a class template. The class template defines an `enum` value `result` which is returned by the function, unless no int values were specified, in which case 0 is returned.

```cpp
template <int ... Ints>
int forwarder()
{
    return computer<Ints ...>::result; // forwarding the Ints
}
```

```cpp
template <> // specialization if no ints are provided
```
int forwarder<>()
{
    return 0;
}

// use as:
cout << forwarder<1, 2, 3>() << ' ' << forwarder<>() << '\n';

The sizeof operator can be used for variadic non-type parameters as well: sizeof...(Ints) would return 3 when used in the first function template for forwarder<1, 2, 3>().

Variadic non-type parameters are used to define variadic literal operators, introduced in section 23.3.

22.6 Tuples

The C++11 standard offers a generalized pair container: the tuple, covered in this section. Before tuples can be used the header file <tuple> must be included.

Whereas std::pair containers have limited functionality and only support two members, tuples have slightly more functionality and may contain an unlimited number of different data types. In that respect a tuple can be considered the ‘template’s answer to C's struct’.

A tuple’s generic declaration (and definition) uses the variadic template notation:

```
template <class ...Types>
class tuple;
```

Here is an example of its use:

```cpp
typedef std::tuple<int, double &, std::string, char const *> tuple_idsc;

double pi = 3.14;
tuple_idsc idsc(59, pi, "hello", "fixed");

// access a field:
std::get<2>(idsc) = "hello world";
```

The std::get<idx>(tupleObject) function template returns a reference to the idxth (zero based) field of the tuple tupleObject. The index is specified as the function template’s non-type template argument.

Tuples may be constructed without specifying initial values. Primitive types are initialized to zeroes; class type fields are initialized by their default constructors. Be aware that in some situations the construction of a tuple may succeed but its use may fail. Consider:

```cpp
tuple<int &> empty;
cout << get<0>(empty);
```

Here the tuple empty cannot be used as its int & field is an undefined reference. However, empty’s construction succeeds.
Tuples may be assigned to each other if their type lists are identical; if supported by their constituent types copy constructors are available as well. Copy construction and assignment is also available if a right-hand type can be converted to its matching left-hand type or if the left-hand type can be constructed from the matching right-hand type. Tuples (matching in number and (convertible) types) can be compared using relational operators as long as their constituent types support comparisons. In this respect tuples are like pairs.

Tuples offer the following static elements (using compile-time initialization):

- std::tuple_size<Tuple>::value returns the number of types defined for the tuple type Tuple. Example:
  
  ```cpp
  cout << tuple_size<tuple_idsc>::value << ‘\n’; // displays: 4
  ```

- std::tuple_element<idx, Tuple>::type returns the type of element idx of Tuple. Example:
  
  ```cpp
  tuple_element<2, tuple_idsc>::type text; // defines std::string text
  ```

The unpack operator can also be used to forward the arguments of a constructor to a tuple data member. Consider a class Wrapper that is defined as a variadic template:

```cpp
template <typename ...Params>
class Wrapper
{
 ...
   public:
   Wrapper(Params &...params);
};
```

This class may be given a tuple data member which should be initialized by the types and values that are used when initializing an object of the class Wrapper using perfect forwarding. Comparable to the way a class may inherit from its template types (cf. section 22.5.3) it may forward its types and constructor arguments to its tuple data member:

```cpp
template <typename ...Params>
class Wrapper
{
   std::tuple<Params...> d_tuple; // same types as used for
                                  // Wrapper itself
   public:
   Wrapper(Params &...params) :
      // initialize d_tuple with
      // Wrapper’s arguments
      d_tuple(std::forward<Params>(params)...) {}
};
```

### 22.7 Computing the return type of function objects

As amply illustrated in chapter 19 function objects play an important role with generic algorithms. Like generic algorithms themselves, function objects can be generically defined as members of class
templates. If the function call operators (operator()) of such classes define parameters then the
types of those parameters may also be abstracted by defining the function call operators themselves
as member templates. Example:

```cpp
template <typename Class>
class Filter
{
    Class obj;
    public:
        template <typename Param>
        Param operator()(Param const &param) const
        {
            return obj(param);
        }
};
```

The template class `Filter` is a wrapper around `Class`, filtering `Class`'s function call operator
through its own function call operator. In the above example the return value of `Class`'s function
call operator is simply passed on, but any other manipulation is of course also possible.

A type that is specified as `Filter`'s template type argument may of course have multiple function
call operators:

```cpp
struct Math
{
    int operator()(int x);
    double operator()(double x);
};
```

Math objects can now be filtered using `Filter<Math> fm` using `Math`'s first or second function
call operator, depending on the actual argument type. With `fm(5)` the `int`-version is used, with
`fm(12.5)` the `double`-version is used.

However, this scheme doesn't work if the function call operators have different return and argument
types. Because of this the following class `Convert` cannot be used with `Filter`:

```cpp
struct Convert
{
    double operator()(int x); // int-to-double
    std::string operator()(double x); // double-to-string
};
```

This problem can be tackled successfully by the class template
`std::result_of<Functor(Typelist)>`. Before using `std::result_of` the header file
`<functional>` must be included.

The `result_of` class template offers a `typedef (type)`, representing the type that is returned by
`Functor<TypeList>`. It can be used as follows to improve the implementation of `Filter`:

```cpp
template <typename Class>
class Filter
{
    Class obj;
};
```
Using this definition, Filter<Convert> fc can be constructed. Now fc(5) returns a double, while fc(4.5) returns a std::string.

The class Convert must define the relationships between its function call operators and their return types. Predefined function objects (like those in the standard template library) already do so, but self-defined function objects must do this explicitly.

If a function object class defines only one function call operator it can define its return type by a typedef. If the above class Convert would only define the first of its two function call operators then the typedef (in the class's public section) should be:

typedef double type;

If multiple function call operators are defined, each with its own signature and return type, then the association between signature and return type is set up as follows (all in the class’s public section):

- define a generic struct result like this:

  template <typename Signature>
  struct result;

- For each function call signature define a specialization of struct result. Convert’s first function call operator gives rise to:

  template <typename Class>
  struct result<Class(int)>
  {
    typedef double type;
  };

  and Convert’s second function call operator to:

  template <typename Class>
  struct result<Class(double)>
  {
    typedef std::string type;
  };

- In cases where function call operators have multiple arguments the specifications should again provide for the correct signatures. A function call operator called with an int and a double, returning a size_t gets:

  template <typename Class>
  struct result<Class(int, double)>
  {
    typedef size_t type;
  };


22.8 Instantiating class templates

Class templates are instantiated when an object of a class template is defined. When a class template object is defined or declared its template parameters must explicitly be specified.

Template parameters are also specified when default template parameter values are specified albeit that in that case the compiler provides the defaults (cf. section 22.4 where double is used as the default type to use for the template's DataType parameter). The actual values or types of template parameters are never deduced from arguments as is done with function template parameters. So to define a Matrix of complex-valued elements, the following syntax is used:

```cpp
Matrix<3, 5, std::complex> complexMatrix;
```

Since the class template Matrix uses a default data type a matrix of double-valued elements can be defined like this:

```cpp
Matrix<3, 5> doubleMatrix;
```

A class template object may be declared using the keyword extern. For example, to declare the matrix complexMatrix use:

```cpp
extern Matrix<3, 5, std::complex> complexMatrix;
```

A class template declaration suffices to compile return values or parameters that are of class template types. Example: the following source file may be compiled, although the compiler hasn't seen the definition of the Matrix class template. Generic classes and (partial) specializations may all be declared. A function expecting or returning a class template object, reference, or parameter automatically becomes a function template itself. This is necessary to allow the compiler to tailor the function to the types of various actual arguments that may be passed to the function:

```cpp
#include <cstddef>

template <size_t Rows, size_t Columns, typename DataType = double>
class Matrix;

template <size_t Columns, typename DataType>
class Matrix<1, Columns, DataType>;

Matrix<1, 12> *function(Matrix<2, 18, size_t> &mat);
```

When class templates are used the compiler must first have seen their implementations. So, template member functions must be known to the compiler when the template is instantiated. This does not mean that all members of a template class are instantiated or must have been seen when a class template object is defined. The compiler only instantiates those members that are actually used. This is illustrated by the following simple class Demo that has two constructors and two members. When we use one constructor and call one member in main note the sizes of the resulting object file and executable program. Next the class definition is modified in that the unused constructor and member are commented out. Again we compile and link the program. Now observe that these latter sizes are identical to the former sizes. There are other ways to illustrate that only used members are instantiated. The nm program could be used. It shows the symbolic contents of object files. Using nm we'll reach the same conclusion: only template member functions that are actually used are instantiated. Here is the class template Demo that was used for our little experiment. In main only
the first constructor and the first member function are called and thus only these members were instantiated:

```cpp
#include <iostream>

template <typename Type>
class Demo
{
    Type d_data;
    public:
        Demo();
        Demo(Type const &value);
        void member1();
        void member2(Type const &value);
};
template <typename Type>
Demo<Type>::Demo()
:  
    d_data(Type())
{}
template <typename Type>
void Demo<Type>::member1()
{
    d_data += d_data;
}

// the following members should be commented out before
// compiling for the 2nd time:

template <typename Type>
Demo<Type>::Demo(Type const &value)
:  
    d_data(value)
{}
template <typename Type>
void Demo<Type>::member2(Type const &value)
{
    d_data += value;
}

int main()
{
    Demo<int> demo;
    demo.member1();
}
```

### 22.9 Processing class templates and instantiations

In section 21.13 the distinction between code depending on template parameters and code not depending on template parameters was introduced. The same distinction also holds true when class templates are defined and used.
Declaration of template friendly functions is verified by the compiler when the template is defined. If a member function in a class template uses a qsort function, then qsort does not depend on a template parameter. Consequently, qsort must be known to the compiler when it encounters qsort's function call. In practice this implies that the `<cstdlib>` header file must have been read by the compiler before it is able to compile the class template definition.

On the other hand, if a template defines a `typename Ret` template type parameter to parameterize the return type of some template member function as in:

```cpp
Ret member();
```

then the compiler may encounter `member` or the class to which `member` belongs in the following locations:

- the location where a class template object is defined. This is called the point of instantiation of the class template object. The compiler must have read the class template's implementation and has performed a basic check for syntactic correctness of member functions like `member`. It won't accept a definition or declaration like `Ret && *member`, because C++ does not support functions returning pointers to rvalue references. Furthermore, it checks whether the actual type name that is used for instantiating the object is valid. This type name must be known to the compiler at the object's point of instantiation.

- the location where the template member function is used. This is called the template member function's point of instantiation. Here the `Ret` parameter must have been specified (or deduced) and at this point `member`'s statements that depend on the `Ret` template parameter are checked for syntactic correctness. For example, if `member` contains a statement like

```cpp
Ret tmp(Ret(), 15);
```

then this is in principle a syntactically valid statement. However, when `Ret = int` the statement fails to compile as `int` does not have a constructor expecting two `int` arguments. Note that this is not a problem when the compiler instantiates an object of `member`'s class. At the point of instantiation of the object its member function `member` is not instantiated and so the invalid `int` construction remains undetected.

### 22.10 Declaring friends

Friend functions are normally constructed as support (free) functions of a class that cannot be implemented and declared as class members. The insertion operator for output streams is a well known example. Friend classes are most often seen in the context of nested classes. Here the inner class declares the outer class as its friend (or the other way around). Again we see a support mechanism: the inner class is constructed to support the outer class.

Like ordinary classes, class templates may declare other functions and classes as their friends. Conversely, ordinary classes may declare template classes as their friends. Here too, the friend is constructed as a special function or class augmenting or supporting the functionality of the declaring class. Although the `friend` keyword can be used by any type of class (ordinary or template) when using class templates the following cases should be distinguished:

- A class template may declare an ordinary function or class as its friend. This is a common friend declaration, such as the insertion operator for `ostream` objects.
• A class template may declare another function template or class template as its friend. In this case, the friend's template parameters may have to be specified. If the actual values of the friend's template parameters must be equal to the template parameters of the class declaring the friend, the friend is said to be a bound friend class or function template. In this case the template parameters of the template specifying the friend declaration determine (bind) the values of the template parameters of the friend class or function. Bound friends result in a one-to-one correspondence between the template's parameters and the friend's template parameters.

• In the most general case a class template may declare another function template or class template to be its friend, irrespective of the friend's actual template arguments. In this case an unbound friend class or function template is declared. The template parameters of the friend class or function template remain to be specified and are not related in some predefined way to the template parameters of the class declaring the friend. If a class template has data members of various types, specified by its template parameters and another class should be allowed direct access to these data members we want to specify any of the current template arguments when specifying such a friend. Rather than specifying multiple bound friends, a single generic (unbound) friend may be declared, specifying the friend's actual template parameters only when this is required.

• The above cases, in which a template is declared as a friend, may also be encountered when ordinary classes are used:
  - The ordinary class declaring ordinary friends has already been covered (chapter 15).
  - The equivalent of bound friends occurs if an ordinary class specifies specific actual template parameters when declaring its friend.
  - The equivalent of unbound friends occurs if an ordinary class declares a generic template as its friend.

22.10.1 Non-templates used as friends in templates

A class template may declare an ordinary function, ordinary member function or ordinary class as its friend. Such a friend may access the class template's private members.

Concrete classes and ordinary functions can be declared as friends, but before a single member function of a class can be declared as a friend, the compiler must have seen the class interface declaring that member. Let's consider the various possibilities:

• A class template may declare an ordinary function to be its friend. It is not completely clear why we would like to declare an ordinary function as a friend. Usually we pass an object of the class declaring the friend to such a function. With class templates this requires us to provide the (friend) function with a template parameter without specifying its types. As the language does not support constructions like

void function(std::vector<Type> &vector)

unless function itself is a template, it is not immediately clear how and why such a friend should be constructed. One reason could be to allow the function access to the class's private static members. In addition such friends could instantiate objects of classes that declare them as their friends. This would allow the friend functions direct access to such object's private members. For example:

    template <typename Type>
22.10. DECLARING FRIENDS

```cpp
class Storage
{
    friend void basic();
    static size_t s_time;
    std::vector<Type> d_data;
public:
    Storage();
};
template <typename Type>
size_t Storage<Type>::s_time = 0;
template <typename Type>
Storage<Type>::Storage()
{};
void basic()
{
    Storage<int>::s_time = time(0);
    Storage<double> storage;
    std::random_shuffle(storage.d_data.begin(), storage.d_data.end());
}

• Declaring an ordinary class to be a class template’s friend probably finds more applications. Here the ordinary (friend) class may instantiate any kind of object of the class template. The friend class may then access all private members of the instantiated class template:

```cpp
template <typename Type>
class Composer
{
    friend class Friend;
    std::vector<Type> d_data;
public:
    Composer();
};
class Friend
{
    Composer<int> d_ints;
public:
    Friend(std::istream &input);
};
inline::Friend::Friend(std::istream &input)
{
    std::copy(std::istream_iterator<int>(input),
              std::istream_iterator<int>(),
              back_inserter(d_ints.d_data));
}

• Alternatively, just a single member function of an ordinary class may be declared as a friend. This requires that the compiler has read the friend class’s interface before declaring the friend. Omitting the required destructor and overloaded assignment operators, the following shows an example of a class whose member randomizer is declared as a friend of the class Composer:

```cpp
template <typename Type>
class Composer;
```
class Friend
{
    Composer<int> *d_ints;
public:
    Friend(std::istream &input);
    void randomizer();
};

template<typename Type>
class Composer
{
    friend void Friend::randomizer();
    std::vector<Type> d_data;
public:
    Composer(std::istream &input)
    {
        std::copy(std::istream_iterator<int>(input),
                  std::istream_iterator<int>(),
                  back_inserter(d_data));
    }
};

inline Friend::Friend(std::istream &input)
:
    d_ints(new Composer<int>(input))
{};

inline void Friend::randomizer()
{
    std::random_shuffle(d_ints->d_data.begin(), d_ints->d_data.end());
}

In this example Friend::d_ints is a pointer member. It cannot be a Composer<int> object as the Composer class interface hasn't yet been seen by the compiler when it reads Friend's class interface. Disregarding this and defining a data member Composer<int> d_ints results in the compiler generating the error

error: field 'd_ints' has incomplete type

'Incomplete type', as the compiler at this points knows of the existence of the class Composer, but as it hasn't seen Composer's interface it doesn't know what size the d_ints data member has.

### 22.10.2 Templates instantiated for specific types as friends

With bound friend class or function templates there is a one-to-one mapping between the template arguments of the friend templates and the template arguments of the class templates declaring them as friends. In this case, the friends themselves are templates too. Here are the various possibilities:

- A function template is a friend of a class template. In this case we don’t experience the problems we encountered with ordinary functions declared as friends of class templates. Since the friend function template itself is a template it may be provided with the required template
arguments allowing it to become the declaring class's friend. The various declarations are organized like this:

- The class template declaring the bound template friend function is defined;
- The (friend) function template is defined, now having access to all the class template's (private) members.

The bound template friend declaration specifies the required template arguments immediately following the template's function name. Without the template argument list affixed to the function name it would remain an ordinary friend function. Here is an example showing a bound friend to create a subset of the entries of a dictionary. For real life examples, a dedicated function object returning `!key1.find(key2)` is probably more useful. For the current example, `operator==` is acceptable:

```cpp
template <typename Key, typename Value>
class Dictionary
{
friend Dictionary<Key, Value>
    subset<Key, Value>(Key const &key,
                   Dictionary<Key, Value> const &dict);

    std::map<Key, Value> d_dict;
public:
    Dictionary();
};

template <typename Key, typename Value>
Dictionary<Key, Value>
subset(Key const &key, Dictionary<Key, Value> const &dict)
{
    Dictionary<Key, Value> ret;

    std::remove_copy_if(dict.d_dict.begin(), dict.d_dict.end(),
                       std::inserter(ret.d_dict, ret.d_dict.begin()),
                       std::bind2nd(std::equal_to<Key>(), key));

    return ret;
}
```

- By declaring a full class template as a class template's friend, all members of the friend class may access all private members of the class declaring the friend. As the friend class only needs to be declared, the organization of the declaration is much easier than when function templates are declared as friends. In the following example a class `Iterator` is declared as a friend of a class `Dictionary`. Thus, the `Iterator` is able to access `Dictionary`'s private data. There are some interesting points to note here:
  - To declare a class template as a friend, that class only needs to be declared as a class template before it is declared as a friend:

```cpp
template <typename Key, typename Value>
class Iterator;

template <typename Key, typename Value>
class Dictionary
{
friend class Iterator<Key, Value>;
}```
– However, members of the friend class may already be used, even though the compiler hasn’t seen the friend class’s interface yet:

```cpp
template <typename Key, typename Value>
template <typename Key2, typename Value2>
Iterator<Key2, Value2> Dictionary<Key, Value>::begin()
{
    return Iterator<Key, Value>(*this);
}
template <typename Key, typename Value>
template <typename Key2, typename Value2>
Iterator<Key2, Value2> Dictionary<Key, Value>::subset(Key const &key)
{
    return Iterator<Key, Value>(*this).subset(key);
}
```

– Of course, the friend class’s interface must eventually be seen by the compiler. Since it’s a support class for Dictionary it can safely define a std::map data member that is initialized by the friend class’s constructor. The constructor may then access the Dictionary’s private data member d_dict:

```cpp
template <typename Key, typename Value>
class Iterator
{
    std::map<Key, Value> &d_dict;

    public:
        Iterator(Dictionary<Key, Value> &dict)
        :
            d_dict(dict.d_dict)
        {}

    – The Iterator member begin may return a map iterator. Since the compiler does not know what the instantiation of the map looks like, a map<Key, Value>::iterator is a template subtype. So it cannot be used as-is, but it must be prefixed by typename (see the function begin’s return type in the next example):

```cpp
template <typename Key, typename Value>
typename std::map<Key, Value>::iterator Iterator<Key, Value>::begin()
{
    return d_dict.begin();
}
```

– In the previous example we might decide that only a Dictionary should be able to construct an Iterator (maybe because we conceptually consider Iterator to be a sub-type of Dictionary). This is easily accomplished by defining Iterator’s constructor in its private section, and by declaring Dictionary to be a friend of Iterator. Consequently, only a Dictionary can create an Iterator. By declaring the constructor of a specific Dictionary type as a friend of Iterator’s we declare a bound friend. This ensures that only that particular type of Dictionary can create Iterators using template parameters identical to its own. Here is how it’s done:

```cpp
template <typename Key, typename Value>
class Iterator
{
    friend Dictionary<Key, Value>::Dictionary();
    std::map<Key, Value> &d_dict;
    Iterator(Dictionary<Key, Value> &dict);
```
public:

In this example, Dictionary's constructor is Iterator's friend. The friend is a template member. Other members can be declared as a class's friend as well. In those cases their prototypes must be used, also specifying the types of their return values. Assuming that

std::vector<Value> sortValues()
is a member of Dictionary then the matching bound friend declaration is:

friend std::vector<Value> Dictionary<Key, Value>::sortValues();

• A class template may define free members, which should have access to the data of instantiations of the class template, but only for matching types. An example is a class template for which the free member operator== must be available. If this is required for a template ClassTemplate, requiring a typename Type template type argument, then a free member

```cpp
template<typename Type>
bool operator==<>(ClassTemplate<Type> const &lhs, ClassTemplate<Type> const &rhs);
```

must have been declared prior to ClassTemplate's interface itself. Within the class interface operator== may then be declared as a friend, specifying operator== as a specialized function template (note the <> following the function name) like this:

```cpp
template<typename Type>
class ClassTemplate
{
  friend bool operator==<>(ClassTemplate<Type> const &lhs, ClassTemplate<Type> const &rhs);
  ...
};
```

Now that the class has been declared, operator=='s implementation may follow.

Finally, the following example can be used as a prototype for situations where bound friends are useful:

```cpp
template <typename T> // a function template
void fun(T *t)
{
  t->not_public();
};
template<typename X> // a free member function template
bool operator==(A<X> const &lhs, A<X> const &rhs);
template <typename X> // a class template
class A
{
  // fun() is used as friend bound to A,
  // instantiated for X, whatever X may be
  friend void fun<A<X>*>(A<X> *);
  // operator== is a friend for A<X> only
  friend bool operator==<>(A<X> const &lhs, A<X> const &rhs);
  int d_data = 10;
};
```
public:
    A();

private:
    void not_public();
};

template <typename X>
A<X>::A()
{
    fun(this);
}

template <typename X>
void A<X>::not_public()
{}

template<typename X> // may access lhs/rhs's private data
bool operator==(A<X> const &lhs, A<X> const &rhs)
{
    return lhs.d_data == rhs.d_data;
}

int main()
{
    A<int> a;

    fun(&a); // fun instantiated for A<int>.
}

22.10.3 Unbound templates as friends

When a friend is declared as an unbound friend it merely declares an existing template to be its friend (no matter how it is instantiated). This may be useful in situations where the friend should be able to instantiate objects of class templates declaring the friend, allowing the friend to access the instantiated object’s private members. Functions, classes, and member functions may all be declared as unbound friends.

Here are the syntactic conventions declaring unbound friends:

- Declaring a function template as an unbound friend: any instantiation of the function template may instantiate objects of the template class and may access its private members. Assume the following function template has been defined

  template <typename Iterator, typename Class, typename Data>
  Class &ForEach(Iterator begin, Iterator end, Class &object,
                  void (Class::*member)(Data &));

  This function template can be declared as an unbound friend in the following class template Vector2:

  template <typename Type>
  class Vector2: public std::vector<std::vector<Type> >
  {
      template <typename Iterator, typename Class, typename Data>
friend Class &ForEach(Iterator begin, Iterator end, Class &object,
    void (Class::*member)(Data &));

... ...

};

If the function template is defined inside some namespace this namespace must be mentioned as well. Assuming that ForEach is defined in the namespace FBB its friend declaration becomes:

template <typename Iterator, typename Class, typename Data>
friend Class &FBB::ForEach(Iterator begin, Iterator end, Class &object,
    void (Class::*member)(Data &));

The following example illustrates the use of an unbound friend. The class Vector2 stores vectors of elements of template type parameter Type. Its process member allows ForEach to call its private rows member. The rows member, in turn, uses another ForEach to call its private columns member. Consequently, Vector2 uses two instantiations of ForEach which is a clear hint for using an unbound friend. It is assumed that Type class objects can be inserted into ostream objects (the definition of the ForEach function template can be found in the cplusplus.yo.zip archive at http://sourceforge.net/projects/cppannotations/). Here is the program:

template <typename Type>
class Vector2: public std::vector<std::vector<Type> >
{
    typedef typename Vector2<Type>::iterator iterator;

    template <typename Iterator, typename Class, typename Data>
friend Class &ForEach(Iterator begin, Iterator end, Class &object,
    void (Class::*member)(Data &));

public:
    void process();

private:
    void rows(std::vector<Type> &row);
    void columns(Type &str);
};

template <typename Type>
void Vector2<Type>::process()
{
    ForEach<iterator, Vector2<Type>, std::vector<Type> > (this->begin(), this->end(), *this, &Vector2<Type>::rows);
}

template <typename Type>
void Vector2<Type>::rows(std::vector<Type> &row)
{
    ForEach(row.begin(), row.end(), *this,
        &Vector2<Type>::columns);
    std::cout << '\n';
}

template <typename Type>
void Vector2<Type>::columns(Type &str)
CHAPTER 22. CLASS TEMPLATES

```cpp
{ std::cout << str << " "; } using namespace std;

int main() {
  Vector2<string> c;
  c.push_back(vector<string>(3, "Hello"));
  c.push_back(vector<string>(2, "World"));
  c.process();
} /*
Generated output:
Hello Hello Hello
World World
*/
```

- Analogously, a full class template may be declared as friend. This allows all instantiations of the friend's member functions to instantiate objects of the class template declaring the friend class. In this case, the class declaring the friend should offer functionality that is useful to different instantiations of its friend class (i.e., instantiations using different template arguments). The syntactic convention is comparable to the convention used when declaring an unbound friend function template:

  ```cpp
template <typename Type>
  class PtrVector
  {
    template <typename Iterator, typename Class>
      friend class Wrapper; // unbound friend class
  };
```

  All members of the class template `Wrapper` may now instantiate `PtrVector`s using any actual type for its `Type` parameter. This allows the `Wrapper` instantiation to access all of `PtrVector`'s private members.

- When only some members of a class template need access to private members of another class template (e.g., the other class template has private constructors and only some members of the first class template need to instantiate objects of the second class template), then the latter class template may declare only those members of the former class template requiring access to its private members as its friends. Again, the friend class's interface may be left unspecified. However, the compiler must be informed that the friend member's class is indeed a class. A forward declaration of that class must therefore be provided. In the next example `PtrVector` declares `Wrapper::begin` as its friend. Note the forward declaration of the class `Wrapper`:

  ```cpp
template <typename Iterator>
  class Wrapper;

  template <typename Type>
  class PtrVector
  { template <typename Iterator> friend
```
22.11. **CLASS TEMPLATE DERIVATION**

```cpp
PtrVector<Type> Wrapper<Iterator>::begin(Iterator const &t1);
...
```

### 22.10.4 Extended friend declarations

The C++11 standard defines *extended friend declarations*. Extended friend declarations are also available for class templates.

Extended friend declarations for class templates allow us to use template type parameters as friend declarations. A template type argument, however, does not necessarily have to be a type for which the keyword `friend` makes sense, like `int`. In those cases the friend declaration is simply ignored.

Consider the following class template, declaring `Friend` as a friend:

```cpp
template <typename Friend>
class Class
{
    friend Friend;
    void msg(); // private, displays some message
};
```

Now, an actual `Friend` class may access all of `Class`’s members:

```cpp
class Concrete
{
    Class<Concrete> d_class;
    Class<std::string> d_string;

    public:
    void msg()
    {
        d_class.msg(); // OK: calls private Class<Concrete>::msg()
        //d_string.msg(); // fails to compile: msg() is private
    }
};
```

A declaration like `Class<int> intClass` is also OK, but here the friend declaration is simply ignored. After all, there are no ‘int members’ to access `Class<int>`’s private members.

### 22.11 Class template derivation

Class templates can be used for inheritance purposes as well. When a class template is used in class derivation, the following situations should be distinguished:

- An existing class template is used as base class when deriving a ordinary class. The derived class itself will partially be a class template, but this is somewhat hidden from view when an object of the derived class is defined.

- An existing class template is used as the base class when deriving another class template. Here the class template characteristics remain clearly visible.
• An ordinary class is used as the base class when deriving a template class. This interesting hybrid allows us to construct class templates that are partially compiled.

These three variants of class template derivation are elaborated in this and the upcoming sections.

Consider the following base class:

```cpp
template<typename T>
class Base
{
    T const &t;

public:
    Base(T const &t);
};
```

The above class is a class template that can be used as a base class for the following derived class template Derived:

```cpp
template<typename T>
class Derived: public Base<T>
{
    public:
        Derived(T const &t);
};
template<typename T>
Derived<T>::Derived(T const &t)
:
    Base(t)
{};
```

Other combinations are also possible. The base class may be instantiated by specifying template arguments, turning the derived class into an ordinary class (showing a class object’s definition as well):

```cpp
class Ordinary: public Base<int>
{
    public:
        Ordinary(int x);
};
inline Ordinary::Ordinary(int x)
:
    Base(x)
{};

Ordinary ordinary(5);
```

This approach allows us to add functionality to a class template, without the need for constructing a derived class template.

Class template derivation pretty much follows the same rules as ordinary class derivation, not involving class templates. Some subtleties that are specific for class template derivation may easily cause confusion like the use of this when members of a template base class are called from a derived class. The reasons for using this are discussed in section 23.1. In the upcoming sections the focus will be on the class derivation proper.
22.11.1 Deriving ordinary classes from class templates

When an existing class template is used as a base class for deriving an ordinary class, the class template parameters are specified when defining the derived class's interface. If in a certain context an existing class template lacks a particular functionality, then it may be useful to derive an ordinary class from a class template. For example, although the class `map` can easily be used in combination with the `find_if()` generic algorithm (section 19.1.16), it requires the construction of a class and at least two additional function objects of that class. If this is considered too much overhead then extending a class template with tailor-made functionality might be considered.

Example: a program executing commands entered at the keyboard might accept all unique initial abbreviations of the commands it defines. E.g., the command `list` might be entered as `l`, `li`, `lis` or `list`. By deriving a class `Handler` from

```
map<string, void (Handler::*)(string const &cmd)>
```

and by defining a member function `process(string const &cmd)` to do the actual command processing a program might simply execute the following `main()` function:

```
int main()
{
    string line;
    Handler cmd;

    while (getline(cin, line))
        cmd.process(line);
}
```

The class `Handler` itself is derived from a `std::map`, in which the map's values are pointers to `Handler`'s member functions, expecting the command line entered by the user. Here are `Handler`'s characteristics:

- The class is derived from a `std::map`, expecting the command associated with each command-processing member as its keys. Since `Handler` uses the map merely to define associations between the commands and the processing member functions and to make available map's typedefs, private derivation is used:

  ```
  class Handler: private std::map<std::string,
      void (Handler::*)(std::string const &cmd)>
  ```

- The actual association can be defined using static private data members: `s_cmds` is an array of `Handler::value_type` values, and `s_cmds_end` is a constant pointer pointing beyond the array's last element:

  ```
  static value_type s_cmds[];
  static value_type* const s_cmds_end;
  ```

- The constructor simply initializes the map from these two static data members. It could be implemented inline:

  ```
  inline Handler::Handler()
    :
      std::map<std::string,
```
• The member `process` iterates along the map's elements. Once the first word on the command line matches the initial characters of the command, the corresponding command is executed. If no such command is found, an error message is issued:

```cpp
void Handler::process(std::string const &line)
{
    istringstream istr(line);
    string cmd;
    istr >> cmd;
    for (iterator it = begin(); it != end(); it++)
    {
        if (it->first.find(cmd) == 0)
        {
            (this->*it->second)(line);
            return;
        }
    }
    cout << "Unknown command: " << line << '\n';
}
```

### 22.11.2 Deriving class templates from class templates

Although it's perfectly acceptable to derive an ordinary class from a class template, the resulting class of course has limited generality compared to its template base class. If generality is important, it's probably a better idea to derive a class template from a class template. This allows us to extend an existing class template with new functionality or to override the functionality of the existing class template.

The class template `SortVector` presented below is derived from the existing class template `std::vector`. It allows us to perform a hierarchic sort of its elements using any ordering of any data members its data elements may contain. To accomplish this there is but one requirement. `SortVector`'s data type must offer dedicated member functions comparing its members.

For example, if `SortVector`'s data type is an object of class `MultiData`, then `MultiData` should implement member functions having the following prototypes for each of its data members which can be compared:

```cpp
bool (MultiData::*)(MultiData const &rhv)
```

So, if `MultiData` has two data members, `int d_value` and `std::string d_text` and both may be used by a hierarchic sort, then `MultiData` should offer the following two members:

```cpp
bool intCmp(MultiData const &rhv); // returns d_value < rhv.d_value
bool textCmp(MultiData const &rhv); // returns d_text < rhv.d_text
```

Furthermore, as a convenience it is assumed that `operator<<` and `operator>>` have been defined for `MultiData` objects.
The class template `SortVector` is directly derived from the template class `std::vector`. Our implementation inherits all members from that base class. It also offers two simple constructors:

```cpp
template <typename Type>
class SortVector: public std::vector<Type>
{
    public:
        SortVector()
        {}
        SortVector(Type const *begin, Type const *end)
        :
            std::vector<Type>(begin, end)
        {}

    Its member `hierarchicSort` is the true *raison d'être* for the class. It defines the hierarchic sort criteria. It expects a pointer to a series of pointers to member functions of the class `Type` as well as a `size_t` representing the size of the array.

The array's first element indicates `Type`'s most significant sort criterion, the array's last element indicates the class's least significant sort criterion. Since the `stable_sort` generic algorithm was designed explicitly to support hierarchic sorting, the member uses this generic algorithm to sort `SortVector`'s elements. With hierarchic sorting, the least significant criterion should be sorted first. `hierarchicSort`'s implementation is therefore easy. It requires a support class `SortWith` whose objects are initialized by the addresses of the member functions passed to the `hierarchicSort()` member:

```cpp
template <typename Type>
void SortVector<Type>::hierarchicSort(
    bool (Type::*arr)(Type const &rhv) const,
    size_t n)
{
    while (n--)
        stable_sort(this->begin(), this->end(), SortWith<Type>(arr[n]));
}
```

The class `SortWith` is a simple wrapper class around a pointer to a predicate function. Since it depends on `SortVector`'s actual data type the class `SortWith` must also be a class template:

```cpp
template <typename Type>
class SortWith
{
    bool (Type::*d_ptr)(Type const &rhv) const;

    SortWith's constructor receives a pointer to a predicate function and initializes the class's `d_ptr` data member:

```cpp
template <typename Type>
SortWith<Type>::SortWith(bool (Type::*ptr)(Type const &rhv) const)
    :
        d_ptr(ptr)
{
}
```

Its binary predicate member (`operator()`) must return `true` if its first argument should eventually be placed ahead of its second argument:
template <typename Type>
bool SortWith<Type>::operator()(Type const &lhv, Type const &rhv) const
{
    return (lhv.*d_ptr)(rhv);
}

The following examples, which can be embedded in a main function, provides an illustration:

- First, A SortVector object is created for MultiData objects. It uses the copy generic algorithm to fill the SortVector object from information appearing at the program’s standard input stream. Having initialized the object its elements are displayed to the standard output stream:

  SortVector<MultiData> sv;
  copy(istream_iterator<MultiData>(cin),
       istream_iterator<MultiData>(),
       back_inserter(sv));

- An array of pointers to members is initialized with the addresses of two member functions. The text comparison is the most significant sort criterion:

  bool (MultiData::*arr[]) (MultiData const &rhv) const =
  {
    &MultiData::textCmp,
    &MultiData::intCmp,
  };

- Next, the array’s elements are sorted and displayed to the standard output stream:

  sv.hierarchicSort(arr, 2);

- Then the two elements of the array of pointers to MultiData’s member functions are swapped, and the previous step is repeated:

  swap(arr[0], arr[1]);
  sv.hierarchicSort(arr, 2);

After compiling the program the following command can be given:

  echo a 1 b 2 a 2 b 1 | a.out

This results in the following output:

  a 1 b 2 a 2 b 1
  ===
  a 1 a 2 b 1 b 2
  ===
  a 1 b 1 a 2 b 2
  ===
22.11.3 Deriving class templates from ordinary classes

An ordinary class may be used as the base class for deriving a template class. The advantage of such an inheritance tree is that the base class's members may all be compiled beforehand. When objects of the class template are now instantiated only the actually used members of the derived class template must be instantiated.

This approach may be used for all class templates having member functions whose implementations do not depend on template parameters. These members may be defined in a separate class which is then used as a base class of the class template derived from it.

As an illustration of this approach we'll develop such a class template below. We'll develop a class Table derived from an ordinary class TableType. The class Table displays elements of some type in a table having a configurable number of columns. The elements are either displayed horizontally (the first \(k\) elements occupy the first row) or vertically (the first \(r\) elements occupy a first column).

When displaying the table's elements they are inserted into a stream. The table is handled by a separate class (TableType), implementing the table's presentation. Since the table's elements are inserted into a stream, the conversion to text (or string) is implemented in Table, but the handling of the strings themselves is left to TableType. We'll cover some characteristics of TableType shortly, concentrating on Table's interface first:

- The class Table is a class template, requiring only one template type parameter: Iterator referring to an iterator to some data type:

  ```cpp
  template <typename Iterator>
  class Table: public TableType
  {
  ```

- Table doesn't need any data members. All data manipulations are performed by TableType.

- Table has two constructors. The constructor's first two parameters are Iterators used to iterate over the elements that must be entered into the table. The constructors require us to specify the number of columns we would like our table to have as well as a FillDirection. FillDirection is an enum, defined by TableType, having values HORIZONTAL and VERTICAL. To allow Table's users to exercise control over headers, footers, captions, horizontal and vertical separators, one constructor has a TableSupport reference parameter. The class TableSupport is developed at a later stage as a virtual class allowing clients to exercise this control. Here are the class's constructors:

  ```cpp
  Table(Iterator const &begin, Iterator const &end,
        size_t nColumns, FillDirection direction);
  Table(Iterator const &begin, Iterator const &end,
        TableSupport &tableSupport,
        size_t nColumns, FillDirection direction);
  ```

- The constructors are Table's only two public members. Both constructors use a base class initializer to initialize their TableType base class and then call the class's private member fill to insert data into the TableType base class object. Here are the constructor's implementations:

  ```cpp
  template <typename Iterator>
  Table<Iterator>::Table(Iterator const &begin, Iterator const &end,
                          TableSupport &tableSupport,
                          size_t nColumns, FillDirection direction)
  ```
CHAPTER 22. CLASS TEMPLATES

TableType(tableSupport, nColumns, direction)
{
    fill(begin, end);
}

template <typename Iterator>
Table<Iterator>::Table(Iterator const &begin, Iterator const &end,
    size_t nColumns, FillDirection direction)
:
    TableType(nColumns, direction)
    {
        fill(begin, end);
    }

• The class's fill member iterates over the range of elements [begin, end), as defined by
  the constructor's first two parameters. As we will see shortly, TableType defines a protected
  data member std::vector<std::string> d_string. One of the requirements of the data
  type to which the iterators point is that this data type can be inserted into streams. So, fill
  uses an ostringstream object to obtain the textual representation of the data, which is then
  appended to d_string:

    template <typename Iterator>
    void Table<Iterator>::fill(Iterator it, Iterator const &end)
    {
        while (it != end)
        {
            std::ostringstream str;
            str << *it++;
            d_string.push_back(str.str());
        }
        init();
    }

This completes the implementation of the class Table. Note that this class template only has three
members, two of them being constructors. Therefore, in most cases only two function templates must
be instantiated: a constructor and the class's fill member. For example, the following defines a
table having four columns, vertically filled by strings extracted from the standard input stream:

    Table<istream_iterator<string> >
    table(istream_iterator<string>(cin), istream_iterator<string>(),
        4, TableType::VERTICAL);

The fill-direction is specified as TableType::VERTICAL. It could also have been specified using
Table, but since Table is a class template its specification would have been slightly more complex:
Table<istream_iterator<string> >::VERTICAL.

Now that the Table derived class has been designed, let's turn our attention to the class TableType.
Here are its essential characteristics:

• It is an ordinary class, designed to operate as Table’s base class.
• It uses various private data members, among which d_colWidth, a vector storing the width
  of the widest element per column and d_indexFun, pointing to the class’s member function
  returning the element in table[row][column], conditional to the table's fill direction.
22.11. CLASS TEMPLATE DERIVATION

TableType also uses a TableSupport pointer and a reference. The constructor not requiring a TableSupport object uses the TableSupport * to allocate a (default) TableSupport object and then uses the TableSupport & as the object's alias. The other constructor initializes the pointer to 0 and uses the reference data member to refer to the TableSupport object provided by its parameter. Alternatively, a static TableSupport object could have been used to initialize the reference data member in the former constructor. The remaining private data members are probably self-explanatory:

```cpp
TableSupport *d_tableSupportPtr;
TableSupport &d_tableSupport;
size_t d_maxWidth;
size_t d_nRows;
size_t d_nColumns;
WidthType d_widthType;
std::vector<size_t> d_colWidth;
size_t (TableType::*d_widthFun)(size_t col) const;
std::string const &(TableType::*d_indexFun)(size_t row, size_t col) const;
```

- The actual string objects populating the table are stored in a protected data member:

  ```cpp
  std::vector<std::string> d_string;
  ```

- The (protected) constructors perform basic tasks: they initialize the object's data members. Here is the constructor expecting a reference to a TableSupport object:

  ```cpp
  #include "tabletype.ih"

  TableType::TableType(TableSupport &tableSupport, size_t nColumns,
                       FillDirection direction)
    :
      d_tableSupportPtr(0),
      d_tableSupport(tableSupport),
      d_maxWidth(0),
      d_nRows(0),
      d_nColumns(nColumns),
      d_widthType(COLUMNWIDTH),
      d_colWidth(nColumns),
      d_widthFun(&TableType::columnWidth),
      d_indexFun(direction == HORIZONTAL ?
                  &TableType::hIndex
                  :
                  &TableType::vIndex)
  {}
  ```

- Once d_string has been filled, the table is initialized by Table::fill. The init protected member resizes d_string so that its size is exactly rows x columns, and it determines the maximum width of the elements per column. Its implementation is straightforward:

  ```cpp
  #include "tabletype.ih"

  void TableType::init()
  {
    if (!d_string.size()) // no elements
```
return; // then do nothing

d_nRows = (d_string.size() + d_nColumns - 1) / d_nColumns;
d_string.resize(d_nRows * d_nColumns); // enforce complete table

// determine max width per column, // and max column width
for (size_t col = 0; col < d_nColumns; col++)
{
    size_t width = 0;
    for (size_t row = 0; row < d_nRows; row++)
    {
        size_t len = stringAt(row, col).length();
        if (width < len)
            width = len;
    }
    d_colWidth[col] = width;

    if (d_maxWidth < width) // max. width so far.
        d_maxWidth = width;
}

• The public member insert is used by the insertion operator (operator<<) to insert a Table into a stream. First it informs the TableSupport object about the table’s dimensions. Next it displays the table, allowing the TableSupport object to write headers, footers and separators:

```
#include "tabletype.ih"

ostream &TableType::insert(ostream &ostr) const
{
    if (!d_nRows)
        return ostr;

    d_tableSupport.setParam(ostr, d_nRows, d_colWidth,
        d_widthType == EQUALWIDTH ? d_maxWidth : 0);

    for (size_t row = 0; row < d_nRows; row++)
    {
        d_tableSupport.hline(row);

        for (size_t col = 0; col < d_nColumns; col++)
        {
            size_t colwidth = width(col);

            d_tableSupport.vline(col);
            ostr << setw(colwidth) << stringAt(row, col);
        }

        d_tableSupport.vline();
    }

d_tableSupport.hline();

    return ostr;
}
```
The cplusplus.yo.zip archive contains TableSupport’s full implementation. This implementation is found in the directory yo/classtemplates/examples/table. Most of its remaining members are private. Among those, these two members return table element \([row][column]\) for, respectively, a horizontally filled table and a vertically filled table:

```cpp
inline std::string const &TableType::hIndex(size_t row, size_t col) const
{
    return d_string[row * d_nColumns + col];
}
inline std::string const &TableType::vIndex(size_t row, size_t col) const
{
    return d_string[col * d_nRows + row];
}
```

The support class TableSupport is used to display headers, footers, captions and separators. It has four virtual members to perform those tasks (and, of course, a virtual constructor):

- **hline(size_t rowIndex)**: called just before displaying the elements in row \(rowIndex\).
- **hline()**: called immediately after displaying the final row.
- **vline(size_t colIndex)**: called just before displaying the element in column \(colIndex\).
- **vline()**: called immediately after displaying all elements in a row.

The reader is referred to the cplusplus.yo.zip archive for the full implementation of the classes Table, TableType and TableSupport. Here is a little program showing their use:

```cpp
/*
   table.cc
*/
#include <fstream>
#include <iostream>
#include <string>
#include <iterator>
#include <sstream>
#include "tablesupport/tablesupport.h"
#include "table/table.h"
using namespace std;
using namespace FBB;

int main(int argc, char **argv)
{
    size_t nCols = 5;
    if (argc > 1)
    {
        istringstream iss(argv[1]);
        iss >> nCols;
    }

    istream_iterator<string> iter(cin); // first iterator isn't const
Table<istream_iterator<string> >
    table(iter, istream_iterator<string>(), nCols,
        argc == 2 ? TableType::VERTICAL : TableType::HORIZONTAL);

    cout << table << '\n';
    return 0;
}
/
Example of generated output:
After: echo a b c d e f g h i j | demo 3
      a e i
      b f j
      c g
      d h
After: echo a b c d e f g h i j | demo 3 h
      a b c
d e f
      g h i
      j
*/

22.12 Static Polymorphism

Chapter 14 introduced polymorphism. Polymorphism allows us to use a base class's interface to call implementations which are defined in derived classes. Traditionally this involves defining Vtables for polymorphic classes, containing pointers to functions that can be overridden in derived classes. Objects of polymorphic classes feature hidden pointers, pointing to their class's Vtables. This type of polymorphism is called dynamic polymorphism, involving late binding as the function to call is determined run-time, rather than compile-time.

In many programs, however, using dynamic polymorphism isn't really required. Usually the derived class objects that are passed to functions expecting base class references are invariants: at fixed locations in programs fixed class types are used to create objects. The polymorphic nature of these objects is used inside the functions that receive these objects, expecting references to their base classes.

The disadvantages of this scheme are that, firstly, inside the functions expecting references to polymorphic base classes execution is somewhat slowed down precisely because of late-binding. Member functions aren't directly called, but are called indirectly via the object's vpointer and their derived class's Vtable. Secondly, programs using dynamic polymorphism tend to become somewhat bloated compared to programs using static binding. The code bloat is caused by the requirement to satisfy at link-time all the references that are mentioned in all the object files comprising the final program. This requirement forces the linker to link all the functions whose addresses are stored in the Vtables of all polymorphic classes, even if these functions are never actually called.

Static polymorphism allows us to avoid these disadvantages. It can be used instead of dynamic polymorphism in cases where the abovementioned invariant holds. Static polymorphism, also called the curiously recurring template pattern, is an example of template meta programming (see also chapter 23 for additional examples of template meta programming).

Whereas dynamic polymorphism is based on the concepts of vpointers, Vtables, and function overriding, static polymorphism capitalizes on the fact that function templates (c.q., member templates) are
only compiled into executable code when they are actually called. This allows us to write code calling functions which are completely non-existent when we write our code. This phenomenon shouldn’t worry us too much. After all, we use a comparable approach when calling a purely virtual function of an abstract base class. The function is really called, but which function will actually be called is determined later in time. With dynamic polymorphism it is determined run-time, with static polymorphism it is determined compile-time.

There’s no need to consider static and dynamic polymorphism as mutually exclusive variants of polymorphism. Rather, both can be used together, combining their strengths.

This section contains several subsections.

- First, the syntax used with static polymorphism is introduced and illustrated;
- Next, it is shown how classes using dynamic polymorphism can be converted into classes that use static polymorphism;
- Finally it is illustrated how static polymorphism can be used to reduce implementation effort. Static polymorphism allows us to implement only once what would have to be implemented repeatedly when only dynamic polymorphism is used.

## 22.12.1 An example of static polymorphism

With static polymorphism a class template takes the role of a base class in dynamic polymorphism. This class template declares several members, which are comparable to the members of a polymorphic base class: they are either support members or they are interfacing to members overridden in derived classes.

With dynamic polymorphism these overriding members are the base class’s virtual members. With static polymorphism there are no virtual members. Instead, the statically polymorphic base class (referred to as the ‘base class’ below) declares a template type parameter (referred to as the ‘derived class type’ below). Next, the base class’s interfacing members call members of the derived class type.

Here is an example of a simple class template which will be used as a base class. Its public interface consists of one member, and, different from dynamic polymorphism, there’s no reference in the class’s interface to any members showing polymorphic behavior:

```cpp
template <class Derived>
struct Base
{
    void interface();
};
```

Let’s have a closer look at the member ‘interface’. This member is called by functions receiving a reference or pointer to the base class, and it calls members that must be available in the derived class type. Before we can call members of the derived class type an object of the derived class type must be available. This object is obtained through inheritance. The derived class type is going to be derived from the base class. Thus Base’s this pointer is also Derived’s this pointer.

When we have a class Derived: public Base then a static_cast<Derived *> can be used to cast a Base * that is actually pointing to a Derived object. A dynamic_cast of course doesn’t apply, as we don’t use dynamic polymorphism. A static_cast is appropriate since our Base * does in fact point to a Derived class object.
Consequently, to call a Derived class member from within interface we can use the following implementation:

```cpp
template<class Derived>
void Base<Derived>::interface()
{
    static_cast<Derived*>(this)->polymorphic();
}
```

Interestingly, when the compiler is given this implementation it cannot determine whether Derived is actually derived from Base. Neither can it determine whether the class Derived indeed offers a member polymorphic. The compiler simply assumes this to be true, in which case the provided implementation is syntactically correct. One of the core characteristics of using templates is that the implementation’s viability is eventually determined at the function’s point of instantiation (cf. section 21.6).

Consequently, in order to use the above scheme we must ensure that derived class types are actually derived from the base class and that the derived class types actually define a member ‘polymorphic’. The first requirement is satisfied using the curiously recurring template pattern:

```cpp
class First: public Base<First>
```

In this curious pattern the class First is derived from Base, which itself is instantiated for First. This is acceptable, as the compiler already has determined that the type First exists. At this point that is all it needs.

The second requirement is satisfied by providing an implementation of the member polymorphic. In chapter 14 we saw that virtual and overriding members belong to the class’s private interface. Applying the same philosophy here, we put polymorphic in First’s private interface, allowing its access by the base class by declaring

```cpp
friend void Base<First>::interface();
```

First’s complete class interface can now be designed, followed by polymorphic’s implementation:

```cpp
class First: public Base<First>
{
    friend void Base<First>::interface();

    private:
        void polymorphic();
};
void First::polymorphic()
{
    std::cout << "polymorphic from First" << std::endl;
}
```

Note that the class First itself is not a class template: its members can be separately compiled and stored in, e.g., a library. Also, as is the case with dynamic polymorphism, the member polymorphic has full access to all of First’s data members and member functions.
Multiple classes can now be designed like `First`, each offering their own implementation of polymorphic. E.g., the member `Second::polymorphic` of the class `Second`, designed like `First`, could be implemented like this:

```cpp
void Second::polymorphic()
{
    std::cout << "polymorphic from Second" << std::endl;
}
```

The polymorphic nature of `Base` becomes apparent once a function template is defined in which `Base::interface` is called. Again, the compiler simply assumes a member `interface` exists when it reads the definition of the following function template:

```cpp
template <class Class>
void fun(Class &object)
{
    object.interface();
}
```

Only at the point where this function is actually called will the compiler verify the viability of the generated code. In the following `main` function a `First` object is passed to `fun`: `First` declares interface through its base class, and `First::polymorphic` is called by `interface`. Then a `Second` object is passed to `fun`, and analogously `Second::polymorphic` is called:

```cpp
int main()
{
    First first;
    fun(first);

    Second second;
    fun(second);
}
```

There are also downsides to using static polymorphism. First, the sentence ‘a `Second` object is passed to `fun`’ isn’t really correct, since `fun` in `fun(first)` and `fun(second)` are different functions, not just calls of one function with different arguments. With static polymorphism every instantiation using its own template parameters results in completely new code once the template (e.g., `fun`) is instantiated. This is something to consider when creating statically polymorphic base classes. If the base class defines data members and member functions, and if these additional members are used by derived class types, then each member will have its own instantiation for each derived class type. This also represents code bloat, albeit of a different kind than observed with dynamic polymorphism. This kind of code bloat can often be somewhat reduced by deriving the base class from its own (ordinary) base class, encapsulating all elements of the statically polymorphic base class that do not depend on its template type parameter.

Second, if different types of statically polymorphic objects are dynamically created then the types of the returned pointers are all different, as are the types of the pointers to their statically polymorphic base classes. These latter pointers are different because they are pointers to `Base<Derived>`, representing different types for different `Derived` types. Consequently, and different from dynamic polymorphism, these pointers cannot be collected in, e.g., a vector of shared pointers to base class pointers. There simply isn’t one base class pointer type. Thus, because of the different base class types, there’s no direct statically polymorphic equivalent to virtual destructors.
Third, as will be illustrated in the next section, designing static polymorphic classes using multiple levels of inheritance is not a trivial task.

Summarizing, static polymorphism is best used in cases where a limited number of different derived class types are defined, where a fixed number of derived class objects are used, and where the statically polymorphic base classes themselves are lean (possibly encapsulating some of their code in ordinary base classes of their own).

### 22.12.2 Converting dynamic polymorphic classes to static polymorphic classes

So you’ve decided to convert some of your dynamically polymorphic classes into statically polymorphic classes. How to realize the conversion?

In chapter 14 the base class `Vehicle` and some derived classes were introduced. `Vehicle`, `Car` and `Truck`'s interfaces look like this (concentrating on the members that are involved in their polymorphic behavior):

```cpp
class Vehicle
{
    size_t d_massFactor;

    public:
        int mass() const;
        int si_mass() const;

    private:
        virtual int vmass() const;
};

class Car: public Vehicle
{
    size_t d_mass;
    private:
        int vmass() const override;
};

class Truck: public Car
{
    size_t d_mass;
    private:
        int vmass() const override;
};
```

Thee members of the user-interface were implemented like this:

```cpp
int Vehicle::mass() const
{
    return d_massFactor * si_mass();
}
int Vehicle::si_mass() const
{
    return vmass();
}
```
When converting dynamically polymorphic classes to statically polymorphic classes we must realize that polymorphic classes serve two functions: first, they define facilities (data members, member functions) that are inherited by derived classes, and second polymorphic classes implement the redefinable interface in a way that fits their purpose.

With statically polymorphic classes it is important that these two aspects are implemented separately. As we've seen in the previous section, a statically polymorphic derived class derives from the base class by using its own class type as argument to the base class's type parameter. This works fine if there's only one level of inheritance.

With multiple levels of inheritance (e.g., Truck -> Car -> Vehicle) Truck's inheritance specification becomes a problem. If it is inheriting from Car, then it implicitly derives from Vehicle<Car>, and thus Car's vmass is called by Vehicle::interface. But Truck must inherit from Car to be able to use Car's features. Additionally inheriting Truck from Vehicle<Truck> isn't an option either, as this creates a class Truck which also inherits from Vehicle<Car> (through Truck's Car base class).

Generally it cannot be ruled out that classes use multiple levels of inheritance. When static polymorphism is used classes must carefully be designed. By separating the redefinable interface from the other elements of the interface, the normal line of inheritance is used to make the protected and public interfaces of (direct or indirect) base classes available to a derived class. The left-hand side of
CHAPTER 22. CLASS TEMPLATES

Figure 22.1 shows this line of inheritance. Each of the left-hand side classes is then used as a base class, together with the static polymorphic base class, to create a statically polymorphic derived class defining its own implementation of the static base class’s redefinable interface. The right-hand side of figure 22.1 shows this part of the design. The XBase classes in the left-hand side of figure 22.1 contain the non-redefinable elements of the original dynamically polymorphic classes. They should of course still be available to the statically polymorphic classes (shown in the right-hand side of figure 22.1). This is realized using friend declarations: the XBase classes declare ‘friend class X’, after which XBase essentially has become part of the class X.

Now that we’ve analyzed the design of statically polymorphic classes using multiple levels of inheritance we’re ready for the conversion of the Vehicle, Car, and Truck dynamically polymorphic classes to statically polymorphic classes. To concentrate on the actual conversion process the design of the classes is kept simple. E.g., usually appearing elements like constructors were not included in the examples. Furthermore, to save some space all implementations of member functions are provided in-class.

First the class Vehicle. Vehicle’s non-redefinable interface is moved to a class VehicleBase, and Vehicle itself is turned into a statically polymorphic base class. Here only Vehicle::d_massFactor needs to be moved to VehicleBase. But in general all members of the original dynamically polymorphic base class that do not use or implement virtual members should be moved to the XBase class. VehicleBase also declares Vehicle to be its friend, so Vehicle is not denied access to its members, now living in VehicleBase. Here is VehicleBase:

```cpp
class VehicleBase
{
    // original Vehicle members, not implementing the
    // redefinable interface

    template <class Derived>
    friend class Vehicle;

    size_t d_massFactor = 1;
};
```

The statically polymorphic Vehicle base class implements the redefinable interface. All other facilities are in VehicleBase, from which it is derived. Here is VehicleBase:

```cpp
template <class Derived>
class Vehicle: public VehicleBase
{
    public:
    int mass() const
    {
        return d_massFactor *
            static_cast<Derived const *>(this)->vmass();
    }
};
```

In this implementation Vehicle does not implement its own vmass member. We cannot define Vehicle<Vehicle>, and with static polymorphism the base class is essentially comparable to an abstract base class. If this is not required, then a default class can be specified for Vehicle’s Derived class, implementing the redefinable interface of the original dynamically polymorphic base class (allowing definitions like Vehicle<> vehicle).
Likewise the class `Car` is split into `CarBase` and `Car`. `CarBase` uses standard inheritance to inherit `VehicleBase`'s protected and public interface. Here is its implementation, adding a private data member for illustration purposes:

```cpp
class CarBase: public VehicleBase
{
    // original Car members, not implementing the
    // redefinable interface
    int x = 1000;
    friend class Car;
};
```

The class `Car` receives `CarBase`'s members through inheritance. Furthermore, it inherits from `Vehicle<Car>`, implementing its own version of `Vehicle`'s redefinable interface. Note the specific friend declaration, allowing `Vehicle<Car>` to access `Car`'s private `vmass` member:

```cpp
class Car: public CarBase, public Vehicle<Car>
{
    friend class Vehicle<Car>;
    int vmass() const
    {
        return x;
    }
};
```

This design pattern can be used for deeper levels of inheritance as well. Once more we need to split a polymorphic class into two separate classes: the polymorphic class `Truck` is split into `TruckBase` and a statically polymorphic class `Truck`, implementing `Vehicle`'s redefinable interface. Here is `TruckBase`, inheriting from `CarBase`, and thus also from `VehicleBase`:

```cpp
class TruckBase: public CarBase
{
    // original Truck members, not implementing the
    // redefinable interface

    friend class Truck;
};
```

Finally, we finish our class conversion for the class `Truck`, now implementing its own version of the statically polymorphic class `Vehicle`:

```cpp
class Truck: public TruckBase, public Vehicle<Truck>
{
    friend class Vehicle<Truck>;

    private:
    int vmass() const
    {
        return 15000;
    }
};
```
The original function `void fun(Vehicle &vehicle)` still needs to be converted to a function template. Here is its definition:

```cpp
template <class Vehicle>
void fun(Vehicle &vehicle)
{
    cout << vehicle.mass() << '\n';
}
```

Note that `Vehicle` is just a formal name. When an object is passed to `fun` it must offer a member `mass`, or compilation will fail. If the object is in fact a `Car` or `Truck`, then their `Vehicle<Type>` static base class member `mass` is called, which in turn uses static polymorphism to call the member `vmass` as implemented by the actually passed class type. The following `main` function displays, respectively, 1000 and 15000:

```cpp
int main()
{
    Cat car;
    fun(car);
    Truck truck;
    fun(truck);
}
```

Realize that this program implements `fun` twice, rather than once in the case of dynamic polymorphism. The same holds true for the `Vehicle` class template: two implementations, one for the `Car` type, and one for the `Truck` type. The statically polymorphic program will be slightly faster, though.

### 22.12.3 Using static polymorphism to avoid reimplementations

Static polymorphism may profitably be used to avoid reimplementing code in an otherwise dynamic polymorphic environment.

Consider the situation where we have a class containing a container of pointers to some polymorphic base class type (like the class `Vehicle` from chapter 14). How to copy such a container to another container? We're not hinting at using shared pointers here, but would like to make a full copy. Clearly, we'll need to duplicate the objects the pointers point at, and assign these new pointers to the copied object's container.

The prototype design pattern is commonly used to create copies of objects of polymorphic classes, given pointers to their base classes.

To apply the prototype design pattern we have to implement `newCopy` in all derived classes. Not by itself a big deal, but static polymorphism can nicely be used here to avoid having to reimplement this function for each derived class.

We start off with an abstract base class `VehicleBase` declaring a pure virtual member `newCopy`:

```cpp
struct VehicleBase
{
    virtual ~VehicleBase();
    VehicleBase *clone() const; // calls newCopy

    virtual ~VehicleBase();
};
```
Next we define a static polymorphic class `CloningVehicle` which is derived from `VehicleBase` (note that we thus combine dynamic and static polymorphism). This class provides the generic implementation of `newCopy`. This is possible because all derived classes can use this implementation. Also, `CloningVehicle` will be re-implemented for each concrete type of vehicle that is eventually used: a `Car`, a `Truck`, an `AmphibiousVehicle`, etc, etc. `CloningVehicle` thus isn’t shared (like `VehicleBase`), but instantiated for each new type of vehicle.

The core characteristic of a statically polymorphic class is that it can use its class template type parameter via a `static_cast` of its own type. A member function like `newCopy` is implemented always the same way, viz., by using the derived class’s copy constructor. Here is the class `CloningVehicle`:

```cpp
template <class Derived>
class CloningVehicle: public VehicleBase
{
  VehicleBase *newCopy() const
  {
    return new Derived(*static_cast<Derived const *>(this));
  }
};
```

And we’re done. All types of vehicles should now be derived from `CloningVehicle`, which automatically provides them with their own implementation of `newCopy`. E.g., a class `Car` looks like this:

```cpp
class Car: public CloningVehicle<Car>
{
  // Car’s interface, no need to either
  // declare or implement newCopy,
  // but a copy constructor is required.
};
```

Having defined a `std::vector<VehicleBase *>` `original` we could create a copy of `original` like this:

```cpp
for(auto pointer: original)
  duplicate.push_back(pointer->clone());
```

Irrespective of the actual type of vehicle to which the pointers point, their `clone` members will return pointers to newly allocated copies of objects of their own types.

### 22.13 Class templates and nesting

When a class is nested within a class template, it automatically becomes a class template itself. The nested class may use the template parameters of the surrounding class, as shown by the follow-
ing skeleton program. Within a class `PtrVector`, a class `iterator` is defined. The nested class receives its information from its surrounding class, a `PtrVector<Type>` class. Since this surrounding class should be the only class constructing its iterators, `iterator`'s constructor is made private and the surrounding class is given access to the private members of `iterator` using a *bound friend* declaration. Here is the initial section of `PtrVector`'s class interface:

```cpp
template <typename Type>
class PtrVector: public std::vector<Type *
```

This shows that the `std::vector` base class stores pointers to `Type` values, rather than the values themselves. Of course, a destructor is now required as the (externally allocated) memory for the `Type` objects must eventually be freed. Alternatively, the allocation might be part of `PtrVector`'s tasks, when it stores new elements. Here it is assumed that `PtrVector`'s clients do the required allocations and that the destructor is implemented later on.

The nested class defines its constructor as a private member, and allows `PtrVector<Type>` objects access to its private members. Therefore only objects of the surrounding `PtrVector<Type>` class type are allowed to construct their `iterator` objects. However, `PtrVector<Type>`'s clients may construct *copies* of the `PtrVector<Type>::iterator` objects they use.

Here is the nested class `iterator`, using a (required) *friend* declaration. Note the use of the `typename` keyword: since `std::vector<Type *>::*begin` depends on a template parameter, it is not yet an instantiated class. Therefore `iterator` becomes an implicit `typename`. The compiler issues a warning if `typename` has been omitted. Here is the class interface:

```cpp
class iterator
{
    friend class PtrVector<Type>;
    typename std::vector<Type *>::iterator d_begin;

    iterator(PtrVector<Type> &vector);

    public:
        Type &operator*();
};
```

The implementation of the members shows that the base class's `begin` member is called to initialize `d_begin`. `PtrVector<Type>::begin`'s return type must again be preceded by `typename`:

```cpp
template <typename Type>
PtrVector<Type>::iterator::iterator(PtrVector<Type> &vector)
    :
        d_begin(vector.std::vector<Type *>::begin())
{};
```

```cpp
template <typename Type>
Type &PtrVector<Type>::iterator::operator*()
{
    return **d_begin;
}
```

The remainder of the class is simple. Omitting all other functions that might be implemented, the function `begin` returns a newly constructed `PtrVector<Type>::iterator` object. It may call the constructor since the class `iterator` declared its surrounding class as its friend:
22.13. CLASS TEMPLATES AND NESTING

```
template <typename Type>
type typename PtrVector<Type>:::iterator PtrVector<Type>:::begin()
{
    return iterator(*this);
}
```

Here is a simple skeleton program, showing how the nested class \texttt{iterator} might be used:

```
int main()
{
    PtrVector<int> vi;
    vi.push_back(new int(1234));
    PtrVector<int>::iterator begin = vi.begin();
    std::cout << *begin << '\n';
}
```

Nested enumerations and nested typedefs can also be defined by class templates. The class \texttt{Table}, mentioned before (section 22.11.3) inherited the enumeration \texttt{TableType::FillDirection}. Had \texttt{Table} been implemented as a full class template, then this enumeration would have been defined in \texttt{Table} itself as:

```
template <typename Iterator>
class Table: public TableType
{
    public:
        enum FillDirection
        {
            HORIZONTAL,  // Typo: should be HORIZONTAL
            VERTICAL
        };

    ...  
};  
```

In this case, the actual value of the template type parameter must be specified when referring to a \texttt{FillDirection} value or to its type. For example (assuming \texttt{iter} and \texttt{nCols} are defined as in section 22.11.3):

```
Table<istream_iterator<string> >:::FillDirection direction =
    argc == 2 ?
        Table<istream_iterator<string> >::VERTICAL
    :
        Table<istream_iterator<string> >::HORIZONTAL;

Table<istream_iterator<string> >
    table(iter, istream_iterator<string>() , nCols, direction);
```
22.14 Constructing iterators

In section 18.2 the iterators used with generic algorithms were introduced. We’ve seen that several types of iterators were distinguished: InputIterators, ForwardIterators, OutputIterators, BidirectionalIterators, and RandomAccessIterators.

To ensure that an object of a class is interpreted as a particular type of iterator, the class must be derived from the class std::iterator. Before a class can be derived from this class the <iterator> header file must be included.

In section 18.2 the characteristics of iterators were discussed. All iterators should support (using Iterator as the generic name of the designed iterator class and Type to represent the (possibly const, in which case the associated operator should be a const member as well) data type to which Iterator objects refer):

- a prefix increment operator (Iterator &operator++();)
- a dereference operator (Type &operator*();)
- a ’pointer to’ operator (Type *operator->();)
- comparison operators testing the (in)equality of two iterator objects (bool operator==(Iterator const &other), bool operator!=(Iterator const &other)).

When iterators are to be used in the context of generic algorithms they must meet additional requirements. This is caused by the fact that generic algorithms perform checks on the types of the iterators they receive. Simple pointers are usually accepted, but if an iterator-object is used it must be able to specify the kind of iterator it represents.

When deriving a class from the class iterator the type of iterator is defined by the class template’s first parameter, and the data type to which the iterator refers is defined by the class template’s second parameter.

The type of iterator that is implemented by the derived class is specified using a so-called iterator_tag, provided as the first template argument of the class iterator. For the five basic iterator types, these tags are:

- std::input_iterator_tag. This tag defines an InputIterator. Iterators of this type allow reading operations, iterating from the first to the last element of the series to which the iterator refers.

  The InputIterator dereference operator should be declared as follows:

  Type const &operator*() const;

  Except for the standard operators there are no further requirements for InputIterators.

- std::output_iterator_tag. This tag defines an OutputIterator. Iterators of this type allow for assignment operations, iterating from the first to the last element of the series to which the iterator refers.

  The OutputIterator dereference operator should allow assignment to the data its dereference operator refers to. Therefore, the OutputIterator dereference operator should be declared as follows:

  Type &operator*();
Except for the standard operators there are no further requirements for OutputIterators.

- **std::forward_iterator_tag**. This tag defines a ForwardIterator. Iterators of this type allow reading and assignment operations, iterating from the first to the last element of the series to which the iterator refers.

  The ForwardIterator dereference operator should allow assignment to the data its dereference operator refers to. Therefore, the ForwardIterator dereference operator should be declared as follows:

  ```
  Type &operator*();
  ```

  Except for the standard operators there are no further requirements for ForwardIterators.

- **std::bidirectional_iterator_tag**. This tag defines a BidirectionalIterator. Iterators of this type allow reading and assignment operations, iterating step by step, possibly in alternating directions, over all elements of the series to which the iterator refers.

  The Bidirectional dereference operator should allow assignment to the data its dereference operator refers to and it should allow stepping backward. BidirectionalIterator should therefore, in addition to the standard operators required for iterators, offer the following operators:

  ```
  Type &operator*();
  Iterator &operator--();
  ```

- **std::random_access_iterator_tag**. This tag defines a RandomAccessIterator. Iterators of this type allow reading and assignment operations, iterating, possibly in alternating directions, over all elements of the series to which the iterator refers using any available (random) stepsize.

  RandomIterator class dereference operators should allow assignment to the data they refer to, and, in addition to the standard operators required for iterators, offer the following operators:

  - `Type &operator*();`, allowing assignment to the data the dereference operator refers to;
  - `Iterator &operator--();`, allowing single steps backward;
  - `Type operator-(Iterator const &rhs) const`, returning the number of data elements between the current and rhs iterator (returning a negative value if rhs refers to a data element beyond the data element this iterator refers to);
  - `Iterator operator+(int step) const`, returning an iterator referring to a data element step data elements beyond the data element this iterator refers to;
  - `Iterator operator-(int step) const`, returning an iterator referring to a data element step data elements before the data element this iterator refers to;
  - `bool operator<(Iterator const &rhs) const`, returning true if the data element this iterator refers to is located before the data element the rhs iterator refers to.

Each iterator tag assumes that a certain set of operators is available. The RandomAccessIterator is the most complex of iterators, as it implies all other iterators.

Note that iterators are always defined over a certain range (`[begin, end)`). Increment and decrement operations may result in undefined behavior of the iterator if the resulting iterator value would refer to a location outside of this range.

Often, iterators only access the elements of the series to which they refer. Internally, an iterator may use an ordinary pointer but it is hardly ever necessary for the iterator to allocate its own memory. Therefore, as the assignment operator and the copy constructor do not have to allocate any memory, their default implementations usually suffice. For the same reason iterators usually don’t require destructors.
Most classes offering members returning iterators do so by having members construct the required iterators that are thereupon returned as objects by those member functions. As the caller of these member functions only has to use or sometimes copy the returned iterator objects, there is usually no need to provide any publicly available constructor, except for the copy constructor. Therefore these constructors are usually defined as private or protected members. To allow an outer class to create iterator objects, the iterator class usually declares the outer class as its friend.

In the following sections the construction of a RandomAccessIterator, the most complex of all iterators, and the construction of a reverse RandomAccessIterator is discussed. The container class for which a random access iterator must be developed may actually store its data elements in many different ways (e.g., using containers or pointers to pointers). Therefore it is difficult to construct a template iterator class which is suitable for a large variety of container classes.

In the following sections the available std::iterator class is used to construct an inner class representing a random access iterator. The reader may follow the approach illustrated there to construct iterator classes for other contexts. An example of such a template iterator class is provided in section 24.7.

The random access iterator developed in the next sections reaches data elements that are only accessible through pointers. The iterator class is designed as an inner class of a class derived from a vector of string pointers.

22.14.1 Implementing a ‘RandomAccessIterator’

In the chapter about containers (chapter 12) it was noted that containers own the information they contain. If they contain objects, then those objects are destroyed once the containers are destroyed. As pointers are not objects their use in containers is discouraged (STL's unique_ptr and shared_ptr type objects may be used, though). Although discouraged, we might be able to use pointer data types in specific contexts. In the following class StringPtr, an ordinary class is derived from the std::vector container that uses std::string * as its data type:

```cpp
#ifndef INCLUDED_STRINGPTR_H_
#define INCLUDED_STRINGPTR_H_

#include <string>
#include <vector>

class StringPtr: public std::vector<std::string *>
{
    public:
        StringPtr(StringPtr const &other);
        ~StringPtr();

        StringPtr &operator=(StringPtr const &other);
    }

#endif
```

This class needs a destructor: as the object stores string pointers, a destructor is required to destroy the strings once the StringPtr object itself is destroyed. Similarly, a copy constructor and overloaded assignment is required. Other members (in particular: constructors) are not explicitly declared here as they are not relevant to this section’s topic.
Assume that we want to be able to use the `sort` generic algorithm with `StringPtr` objects. This algorithm (see section 19.1.58) requires two `RandomAccessIterators`. Although these iterators are available (via `std::vector`'s `begin` and `end` members), they return iterators to `std::string *s`, which cannot sensibly be compared.

To remedy this, we may define an internal type `StringPtr::iterator`, not returning iterators to pointers, but iterators to the objects these pointers point to. Once this `iterator` type is available, we can add the following members to our `StringPtr` class interface, hiding the identically named, but useless members of its base class:

```cpp
StringPtr::iterator begin(); // returns iterator to the first element
StringPtr::iterator end();  // returns iterator beyond the last element
```

Since these two members return the (proper) iterators, the elements in a `StringPtr` object can easily be sorted:

```cpp
int main()
{
    StringPtr sp; // assume sp is somehow filled
    sort(sp.begin(), sp.end()); // sp is now sorted
}
```

To make this all work, the type `StringPtr::iterator` must be defined. As suggested by its type name, `iterator` is a nested type of `StringPtr`. To use a `StringPtr::iterator` in combination with the `sort` generic algorithm it must also be a `RandomAccessIterator`. Therefore, `StringPtr::iterator` itself must be derived from the existing class `std::iterator`.

To derive a class from `std::iterator`, both the iterator type and the data type the iterator points to must be specified. Caveat: our iterator takes care of the `string *` dereferencing; so the required data type is `std::string`, and not `std::string *`. The class `iterator` therefore starts its interface as:

```cpp
class iterator:
    public std::iterator<std::random_access_iterator_tag, std::string>
```

Since its base class specification is quite complex, we could consider associating this type with a shorter name using the following `typedef`:

```cpp
typedef std::iterator<std::random_access_iterator_tag, std::string>
    Iterator;
```

In practical situations, if the type (`Iterator`) is used only once or twice, the type definition only adds clutter to the interface, and is better not used.

Now we’re ready to redesign `StringPtr`’s class interface. It offers members returning (reverse) iterators, and a nested `iterator` class. Here is its interface:

```cpp
class StringPtr: public std::vector<std::string *>
{
    public:
```
class iterator: public
    std::iterator<std::random_access_iterator_tag, std::string>
{
    friend class StringPtr;
    std::vector<std::string *>::iterator d_current;

    iterator(std::vector<std::string *>::iterator const &current);

    public:
    iterator &operator--();
    iterator operator--(int);
    iterator &operator++();
    iterator operator++(int);
    bool operator==(iterator const &other) const;
    bool operator!=(iterator const &other) const;
    int operator-(iterator const &rhs) const;
    std::string &operator*() const;
    bool operator<(iterator const &other) const;
    iterator operator+(int step) const;
    iterator operator-(int step) const;
    iterator &operator+=(int step); // increment over ‘n’ steps
    iterator &operator-=(int step); // decrement over ‘n’ steps
    std::string *operator->() const;// access the fields of the
    // struct an iterator points
    // to. E.g., it->length()
};

typedef std::reverse_iterator<iterator> reverse_iterator;

iterator begin();
iterator end();
reverse_iterator rbegin();
reverse_iterator rend();
};

As usual, the interface offers hooks for a more detailed study of the class.

First we have a look at StringPtr::iterator’s characteristics:

- **iterator defines StringPtr as its friend, so iterator’s constructor may remain private.** Only the StringPtr class itself is now able to construct iterators, which seems like a sensible thing to do. Under the current implementation, *copy-construction* should of course also be possible. Furthermore, since an iterator is already provided by StringPtr’s base class, we can use that iterator to access the information stored in the StringPtr object.

- **StringPtr::begin and StringPtr::end may simply return iterator objects. They are implementated like this:**

```cpp
inline StringPtr::iterator StringPtr::begin()
{
    return iterator(this->std::vector<std::string *>::begin());
}
inline StringPtr::iterator StringPtr::end()
{
    return iterator(this->std::vector<std::string *>::end());
}
```
• All of iterator’s remaining members are public. It’s very easy to implement them, mainly manipulating and dereferencing the available iterator d_current. A RandomAccessIterator (which is the most complex of iterators) requires a series of operators. They usually have very simple implementations, making them good candidates for inline-members:

- iterator &operator++(); the pre-increment operator:
  
  inline StringPtr::iterator &StringPtr::iterator::operator++()
  {
    ++d_current;
    return *this;
  }

- iterator operator++(int); the post-increment operator:
  
  inline StringPtr::iterator StringPtr::iterator::operator++(int)
  {
    return iterator(d_current++);
  }

- iterator &operator--(); the pre-decrement operator:
  
  inline StringPtr::iterator &StringPtr::iterator::operator--()
  {
    --d_current;
    return *this;
  }

- iterator operator--(int); the post-decrement operator:
  
  inline StringPtr::iterator StringPtr::iterator::operator--(int)
  {
    return iterator(d_current--);
  }

- iterator &operator=(iterator const &other); the overloaded assignment operator. Since iterator objects do not allocate any memory themselves, the default assignment operator can be used.

- bool operator==(iterator const &rhv) const; testing the equality of two iterator objects:
  
  inline bool StringPtr::iterator::operator==(iterator const &other) const
  {
    return d_current == other.d_current;
  }

- bool operator<(iterator const &rhv) const; testing whether the left-hand side iterator points to an element of the series located before the element pointed to by the right-hand side iterator:
  
  inline bool StringPtr::iterator::operator<(iterator const &other) const
  {
    return d_current < other.d_current;
  }

- int operator-(iterator const &rhv) const; returning the number of elements between the element pointed to by the left-hand side iterator and the right-hand side
iterator (i.e., the value to add to the left-hand side iterator to make it equal to the value of the right-hand side iterator):

```cpp
inline int StringPtr::iterator::operator-(iterator const &rhs) const
{
    return d_current - rhs.d_current;
}
```

- `Type &operator*() const;` returning a reference to the object to which the current iterator points. With an `InputIterator` and with all `const_iterators`, the return type of this overloaded operator should be `Type const &`. This operator returns a reference to a string. This string is obtained by dereferencing the dereferenced `d_current` value. As `d_current` is an iterator to `string *` elements, two dereference operations are required to reach the string itself:

```cpp
inline std::string &StringPtr::iterator::operator*() const
{
    return **d_current;
}
```

- `iterator operator+(int stepsize) const;` this operator advances the current iterator by stepsize:

```cpp
inline StringPtr::iterator StringPtr::iterator::operator+(int step) const
{
    return iterator(d_current + step);
}
```

- `iterator operator-(int stepsize) const;` this operator decreases the current iterator by stepsize:

```cpp
inline StringPtr::iterator StringPtr::iterator::operator-(int step) const
{
    return iterator(d_current - step);
}
```

- `iterator(iterator const &other);` iterators may be constructed from existing iterators. This constructor doesn’t have to be implemented, as the default copy constructor can be used.

- `std::string *operator->() const` is an additionally added operator. Here only one dereference operation is required, returning a pointer to the string, allowing us to access the members of a string via its pointer:

```cpp
inline std::string *StringPtr::iterator::operator->() const
{
    return *d_current;
}
```

- Two more additionally added operators are `operator+=` and `operator-=`. They are not formally required by `RandomAccessIterators`, but they come in handy anyway:

```cpp
inline StringPtr::iterator &StringPtr::iterator::operator+=(int step)
{
    d_current += step;
    return *this;
}
```

```cpp
inline StringPtr::iterator &StringPtr::iterator::operator-=(int step)
{
    d_current -= step;
    return *this;
}
```
The interfaces required for other iterator types are simpler, requiring only a subset of the interface required by a random access iterator. E.g., the forward iterator is never decremented and never incremented over arbitrary step sizes. Consequently, in that case all decrement operators and 
operator+(int step) can be omitted from the interface. Of course, the tag to use would then be 
std::forward_iterator_tag. The tags (and the set of required operators) vary accordingly for 
the other iterator types.

### 22.14.2 Implementing a ‘reverse_iterator’

Once we've implemented an iterator, the matching reverse iterator can be implemented in a jiffy. Comparable to the std::iterator a std::reverse_iterator exists, that nicely implements the reverse iterator for us once we have defined an iterator class. Its constructor merely requires an object of the iterator type for which we want to construct a reverse iterator.

To implement a reverse iterator for StringPtr we only need to define the reverse_iterator type in its interface. This requires us to specify only one line of code, which must be inserted after the interface of the class iterator:

```cpp
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Also, the well known members rbegin and rend are added to StringPtr's interface. Again, they can easily be implemented inline:

```cpp
inline StringPtr::reverse_iterator StringPtr::rbegin()
{
    return reverse_iterator(end());
}
inline StringPtr::reverse_iterator StringPtr::rend()
{
    return reverse_iterator(begin());
}
```

Note the arguments the reverse_iterator constructors receive: the begin point of the reversed iterator is obtained by providing reverse_iterator's constructor with the value returned by the member end: the endpoint of the normal iterator range; the endpoint of the reversed iterator is obtained by providing reverse_iterator's constructor with the value returned by the member begin: the begin point of the normal iterator range.

The following small program illustrates the use of StringPtr's RandomAccessIterator:

```cpp
#include <iostream>
#include <algorithm>
#include "stringptr.h"
using namespace std;

int main(int argc, char **argv)
{
    StringPtr sp;

    while (*argv)
        sp.push_back(new string(*argv++));
```
sort(sp.begin(), sp.end());
copy(sp.begin(), sp.end(), ostream_iterator<string>(cout, " "));

cout << "\n======\n";
sort(sp.rbegin(), sp.rend());
copy(sp.begin(), sp.end(), ostream_iterator<string>(cout, " "));
cout << '\n';
} /*
   when called as:
   a.out bravo mike charlie zulu quebec

generated output:
   a.out bravo charlie mike quebec zulu
   ======
   zulu quebec mike charlie bravo a.out
*/

Although it is thus possible to construct a reverse iterator from a normal iterator, the opposite does not hold true: it is not possible to initialize a normal iterator from a reverse iterator.

Assume we would like to process all lines stored in `vector<string>` `lines` up to any trailing empty lines (or lines only containing blanks) it might contain. How should we proceed? One approach is to start the processing from the first line in the vector, continuing until the first of the trailing empty lines. However, once we encounter an empty line it does of course not have to be the first line of the set of trailing empty lines. In that case, we’d better use the following algorithm:

- First, use

  \[
  \text{rit} = \text{find\_if}(\text{lines.rbegin()}, \text{lines.rend()}, \text{NonEmpty}());
  \]

  to obtain a reverse_iterator `rit` pointing to the last non-empty line.

- Next, use

  \[
  \text{for\_each}(\text{lines.begin()}, --\text{rit}, \text{Process}());
  \]

  to process all lines up to the first empty line.

However, we can’t mix iterators and reverse iterators when using generic algorithms. So how can we initialize the second iterator using the available reverse_iterator? The solution is not very difficult, as an iterator may be initialized from a pointer. Although the reverse iterator `rit` is not a pointer, `&*(\text{rit} - 1)` or `&*-\text{rit}` is. So we use

\[
\text{for\_each}(\text{lines.begin()}, &*--\text{rit}, \text{Process}());
\]

to process all the lines up to the first of the set of trailing empty lines. In general, if `rit` is a reverse_iterator pointing to some element and we need an iterator to point to that element, we may use `&*rit` to initialize the iterator. Here, the dereference operator is applied to reach the element the reverse iterator refers to. Then the address operator is applied to obtain its address with which we can initialize the iterator.
When defining a const_reverse_iterator (e.g., matching a const_iterator class), then the const_iterator’s operator* member should be a member returning a non-modifiable value or object. Since a const_reverse_iterator uses the iterator’s operator- member, we’re running against a small conceptual conflict. On the one hand, a std::input_iterator_tag is inappropriate, since we must allow decrementing the iterator. On the other hand, a std::bidirectional_iterator is inappropriate, since we don’t allow modification of the data.

Iterator tags are primarily conceptual. If const_iterators and const_reverse_iterators only allow increment operations, then an input_iterator_tag most closely matches the iterator’s intended use. Hence this tag is used below.

Furthermore, in line with the nature of a input_iterator_tag our const_iterator should not offer an operator-. This, of course, causes problems: a reverse iterator must be able to use the iterator’s operator- member. This can easily be solved by stashing the iterator’s operator- in the iterator’s private section, and declaring std::reverse_iterator< iterator> its friend (note that declaring a (const_)reverse_iterator that is derived from std::reverse_iterator doesn’t solve the issue: it is std::reverse_iterator that calls the iterator’s operator-, not a class that is derived from it).

There is, however, another issue. After deriving a const_reverse_iterator from const_iterator, and subsequently dereferencing a const_reverse_iterator, the compiler generates an error message like the following (using Type = int):

```
error: invalid initialization of non-const reference of type 'std::reverse_iterator< const_iterator >::reference {aka int&}' from an rvalue of type 'int'
  return *--__tmp;
```

This message is caused by std::reverse_iterator by default expecting that the iterator’s operator* returns a reference to a modifiable Type.

To control such default expectancies, iterators can use typedefs to fine-tune these expectancies. The following typedefs are interpreted by std::reverse_iterator:

- `pointer` - the type of the pointer to the data (e.g., Type *)
- `const_pointer` - the type of a pointer to immutable data (e.g., Type const *)
- `reference` - the type of a reference to the data (e.g., Type &)
- `const_reference` - the type of a reference to immutable data (e.g., Type const &)
- `difference_type` - the type representing differences between pointers (by default ‘ptrdiff_t’)

The mentioned error can simply be prevented by declaring ‘typedef int const &reference’ in the iterator class. Alternatively, int const may be specified as the data type for the std::iterator’s base class.

To define a iterator, const_iterator, reverse_iterator and const_reverse_iterator for a class Data the following framework can be used:

```cpp
#include <string>
#include <iterator>
```
class Data
{
    std::string *d_data;
    size_t d_n;

    public:
    class iterator;
    class const_iterator;

    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
};

class Data::iterator: public std::iterator<std::input_iterator_tag,
    std::string>
{
    public:
    iterator() = default;

    iterator &operator++();
    std::string &operator*();

    private:
    friend class Data;
    iterator(std::string *data, size_t idx);

    friend class std::reverse_iterator<iterator>;
    iterator &operator--();
};

bool operator==(Data::iterator const &lhs, Data::iterator const &rhs);

class Data::const_iterator: public std::iterator<std::input_iterator_tag,
    std::string const>
{
    public:
    const_iterator() = default;

    const_iterator &operator++();
    std::string const &operator*() const;

    private:
    friend class Data;
    const_iterator(std::string const *data, size_t idx);

    friend class std::reverse_iterator<const_iterator>;
    const_iterator &operator--();
};

bool operator==(Data::const_iterator const &lhs, Data::const_iterator const &rhs);
int main()
{
    Data::iterator iter;
    Data::reverse_iterator riter(iter);
    *riter;

    Data::const_iterator citer;
    Data::const_reverse_iterator criter(citer);
    *criter;
}

Chapter 23

Advanced Template Use

The main purpose of templates is to provide a generic definition of classes and functions that may then be tailored to specific types.

But templates allow us to do more than that. If not for compiler implementation limitations, templates could be used to program, at compile-time, just about anything we use computers for. This remarkable feat, offered by no other current-day computer language, stems from the fact that templates allow us to do three things at compile-time:

- Templates allow us to do integer arithmetic (and to save computed values symbolically);
- Templates allow us to make compile-time decisions;
- Templates allow us to do things repeatedly.

Of course, asking the compiler to compute, e.g., prime numbers, is one thing. But it’s a completely different thing to do so in an award winning way. Don’t expect speed records to be broken when the compiler performs complex calculations for us. But that’s all beside the point. In the end we can ask the compiler to compute virtually anything using C++’s template language, including prime numbers....

In this chapter these remarkable features of templates are discussed. Following a short overview of subtleties related to templates the main characteristics of template meta programming are introduced.

In addition to template type and template non-type parameters there is a third kind of template parameter, the template template parameter. This kind of template parameter is introduced next, laying the groundwork for the discussion of trait classes and policy classes.

This chapter ends with the discussion of several additional and interesting applications of templates: adapting compiler error messages, conversions to class types and an elaborate example discussing compile-time list processing.

Much of the inspiration for this chapter came from two highly recommended books: Andrei Alexandrescu’s 2001 book Modern C++ design (Addison-Wesley) and Nicolai Josutis and David Vandevoorde’s 2003 book Templates (Addison-Wesley).
23.1 Subtleties

In section 22.2.1 a special application of the keyword typename was discussed. There we learned that it is not only used to define a name for a (complex) type, but also to distinguish types defined by class templates from members defined by class templates. In this section two more applications of typename are introduced:

- In section 23.1.1 we apply typename to situations where types nested in templates are returned from member functions of class templates;
- in section 23.1.2 we cover the problem of how to refer to base class templates from derived class templates.

In addition to the special applications of typename section 23.1.3 introduces some new syntax that is related to the extended use of the keyword typename: ::template, .template and ->template are used to inform the compiler that a name used inside a template is itself a class template.

23.1.1 Returning types nested under class templates

In the following example a nested class, not depending on a template parameter, is defined inside a class template. The class template member nested returns an object of this nested class. The example uses a (deprecated) in-class member implementation. The reason for this shortly becomes clear.

```cpp
template <typename T>
class Outer
{
    public:
    class Nested
    {
    };
    Nested nested() const
    {
        return Nested();
    }
};
```

The above example compiles flawlessly. Inside the class Outer there is no ambiguity with respect to the meaning of nested’s return type.

However, following good practices inline and template members should be implemented below their class interfaces (see section 7.9.1). So we remove the implementation from the interface and put it below the interface:

```cpp
template <typename T>
class Outer
{
    public:
    class Nested
    {
    };

    Nested nested() const;
};
```
23.1. SUBTLETIES

Suddenly the compiler refuses to compile the `nested` member, producing an error message like

```
error: expected constructor, destructor, or type conversion before 'Outer'.
```

Now that the implementation is moved out of the interface the return type (i.e., `Outer<T>::Nested`) refers to a type defined by `Outer<T>` rather than to a member of `Outer<T>`. Here `typename` must once again be used. The general rule being: the keyword `typename` must be used whenever a type is referred to that is a subtype of a type that itself depends on a template type parameter. When using the inline implementation no such dependency is used as the function's return type is simply `Nested`. When implementing the function outside of the class interface (which should be considered 'good practice') then a specification of the class defining `Nested` must be provided for the function's return type. So it becomes `Outer<T>::Nested` which clearly is a type depending on a template type parameter.

Like before, writing `typename` in front of `Outer<T>::Nested` removes the compilation error. Thus, the correct implementation of the function `nested` becomes:

```
template <typename T>
typename Outer<T>::Nested Outer<T>::nested() const
{
    return Nested();
}
```

### 23.1.2 Type resolution for base class members

Below we see two class templates, `Base` and `Derived`, `Base` being `Derived`'s base class:

```cpp
#include <iostream>

template <typename T>
class Base
{
    public:
        void member();
};
template <typename T>
void Base<T>::member()
{
    std::cout << "This is Base<T>::member()\n";
}
template <typename T>
class Derived: public Base<T>
{
```
CHAPTER 23. ADVANCED TEMPLATE USE

```cpp
public:
    Derived();
};
template <typename T>
Derived<T>::Derived()
{
    member();
}
```

This example won’t compile, and the compiler tells us something like:

```
error: there are no arguments to 'member' that depend on a template parameter, so a declaration of 'member' must be available
```

This error causes some confusion as ordinary (non-template) base classes readily make their public and protected members available to classes that are derived from them. This is no different for class templates, but only if the compiler can figure out what we mean. In the above example the compiler can’t as it doesn’t know for what type `T` the member function `member` must be initialized when called from `Derived<T>::Derived`.

To appreciate why this is true, consider the situation where we have defined a specialization:

```cpp
template <>
Base<int>::member()
{
    std::cout << "This is the int-specialization\n";
}
```

Since the compiler, when `Derived<SomeType>::Derived` is called, does not know whether a specialization of `member` is in effect, it can’t decide (when compiling `Derived<T>::Derived`) for what type to instantiate `member`. It can’t decide this when compiling `Derived<T>::Derived` as `member`’s call in `Derived::Derived` doesn’t require a template type parameter.

In cases like these, where no template type parameter is available to determine which type to use, the compiler must be told that it should postpone its decision about the template type parameter to use (and therefore about the particular (here: `member`) function to call) until instantiation time.

This may be implemented in two ways: either by using `this` or by explicitly mentioning the base class, instantiated for the derived class’s template type(s). When `this` is used the compiler is informed that we’re referring to the type `T` for which the template was instantiated. Any confusion about which member function to use (the derived class or base class member) is resolved in favor of the derived class member. Alternatively, the base or derived class can explicitly be mentioned (using `Base<T>` or `Derived<T>`) as shown in the next example. Note that with the int template type the int specialization is used.

```cpp
#include <iostream>
template <typename T>
class Base
{
    public:
        virtual void member();
};
template <typename T>
```
```cpp
void Base<T>::member()
{
    std::cout << "This is Base<T>::member()\n";
}
template <>
void Base<int>::member()
{
    std::cout << "This is the int-specialization\n";
}
template <typename T>
class Derived: public Base<T>
{
    public:
        Derived();
        virtual void member();
};
template <typename T>
void Derived<T>::member()
{
    std::cout << "This is Derived<T>::member()\n";
}
template <typename T>
Derived<T>::Derived()
{
    this->member(); // Using 'this' implies using the
                    // type for which T was instantiated
        Derived<T>::member(); // Same: calls the Derived member
        Base<T>::member();    // Same: calls the Base member
    std::cout << "Derived<T>::Derived() completed\n";
}
int main()
{
    Derived<double> d;
    Derived<int> i;
}
```

Generated output:
This is Derived<T>::member()
This is Derived<T>::member()
This is Base<T>::member()
Derived<T>::Derived() completed
This is Derived<T>::member()
This is Derived<T>::member()
This is the int-specialization
Derived<T>::Derived() completed
*/

The above example also illustrates the use of virtual member templates (although virtual member templates aren’t often used). In the example `Base` declares a virtual void member and `Derived` defines its overriding function `member`. In that case `this->member()` in `Derived::Derived` calls, due to `member`'s virtual nature, `Derived::member`. The statement `Base<T>::member()`, however, always calls `Base`'s `member` function and can be used to bypass dynamic polymorphism.
23.1.3 ::template, .template and ->template

In general, the compiler is able to determine the true nature of a name. As discussed, this is not always the case and sometimes we have to advise the compiler. The typename keyword may often be used for that purpose.

But typename cannot always come to the rescue. While parsing a source the compiler receives a series of tokens, representing meaningful units of text encountered in the program’s source. A token could represent, e.g., an identifier or a number. Other tokens represent operators, like =, + or <. It is precisely the last token that may cause problems as it may have very different meanings. The correct meaning cannot always be determined from the context in which the compiler encounters <. In some situations the compiler does know that < does not represent the less than operator, as when a template parameter list follows the keyword template, e.g.,

```
template <typename T, int N>
```

Clearly, in this case < does not represent a ‘less than’ operator.

The special meaning of < when it is preceded by template forms the basis for the syntactic constructs discussed in this section.

Assume the following class has been defined:

```
template <typename Type>
class Outer
{
    public:
        template <typename InType>
class Inner
        {
            public:
                template <typename X>
                void nested();
            }
    }
}
```

The class template Outer defines a nested class template Inner. Inner in turn defines a template member function.

Next a class template Usage is defined, offering a member function caller expecting an object of the above Inner type. An initial setup for Usage looks like this:

```
template <typename T1, typename T2>
class Usage
{
    public:
        void caller(Outer<T1>::Inner<T2> &obj);
    ...
}
```

The compiler won’t accept this as it interprets Outer<T1>::Inner as a class type. But there is no class Outer<T1>::Inner. Here the compiler generates an error like:

```
error: ‘class Outer<T1>::Inner’ is not a type
```
To inform the compiler that `Inner` itself is a template, using the template type parameter `<T2>`, the `::template` construction is required. It tells the compiler that the next `<` should not be interpreted as a ‘less than’ token, but rather as a template type argument. So, the declaration is modified to:

```cpp
void caller(Outer<T1>::template Inner<T2> &obj);
```

This still doesn’t get us where we want to be: after all `Inner<T2>` is a type, nested under a class template, depending on a template type parameter. In fact, the original `Outer<T1>::Inner<T2> &obj` declaration results in a series of error messages, one of them looking like this:

```cpp
error: expected type-name before ‘&’ token
```

As is often the case this error message nicely indicates what should be done to get it right: add `typename`:

```cpp
void caller(typename Outer<T1>::template Inner<T2> &obj);
```

Of course, `caller` itself is not only just declared, it must also be implemented. Assume that its implementation should call `Inner`'s member `nested`, instantiated for yet another type `X`. The class template `Usage` should therefore receive a third template type parameter, called `T3`. Assume it has been defined. To implement `caller`, we write:

```cpp
void caller(typename Outer<T1>::template Inner<T2> &obj) {
    obj.nested<T3>();
}
```

Once again we run into a problem. In the function’s body the compiler once again interprets `<` as ‘less than’, seeing a logical expression having as its right-hand side a primary expression instead of a function call specifying a template type `T3`.

To tell the compiler that is should interpret `<T3>` as a type to instantiate, the `template` keyword must once again be used. This time it is used in the context of the member selection operator. We write `.template` to inform the compiler that what follows is not a ‘less than’ operator, but rather a type specification. The function’s final implementation becomes:

```cpp
void caller(typename Outer<T1>::template Inner<T2> &obj) {
    obj.template nested<T3>();
}
```

Instead of defining value or reference parameters functions may also define pointer parameters. Had `obj` been defined as a pointer parameter the implementation would have had to use the `->template` construction, rather than the `.template` construction. E.g.,

```cpp
void caller(typename Outer<T1>::template Inner<T2> *ptr) {
    ptr->template nested<T3>();
}
```
23.2 Template Meta Programming

23.2.1 Values according to templates

In template programming values are preferably represented by *enum* values. Enums are preferred over, e.g., int const values since enums never require any linkage. They are pure symbolic values with no memory representation whatsoever.

Consider the situation where a programmer must use a cast, say a reinterpret_cast. A problem with a reinterpret_cast is that it is the ultimate way to turn off all compiler checks. All bets are off, and we can write extreme but absolutely pointless reinterpret_cast statements, like

```cpp
int intVar = 12;
ostream &ostr = reinterpret_cast<ostream &>(intVar);
```

Wouldn’t it be nice if the compiler would warn us against such oddities by generating an error message?

If that’s what we’d like the compiler to do, there must be some way to distinguish madness from weirdness. Let’s assume we agree on the following distinction: reinterpret casts are never acceptable if the target type represents a larger type than the expression (source) type, since that would immediately result in exceeding the amount of memory that’s actually available to the target type. For this reason it’s clearly silly to reinterpret_cast<double *>(&intVar), but reinterpret_cast<char *>(&intVar) could be defensible.

The intent is now to create a new kind of cast, let’s call it reinterpret_to_smaller_cast. It should only be allowed to perform a reinterpret_to_smaller_cast if the target type occupies less memory than the source type (note that this exactly the opposite reasoning as used by Alexandrescu (2001), section 2.1).

To start, we construct the following template:

```cpp
template<typename Target, typename Source>
Target &reinterpret_to_smaller_cast(Source &source)
{
    // determine whether Target is smaller than source
    return reinterpret_cast<Target &>(source);
}
```

At the comment an enum-definition is inserted defining a symbol having a suggestive name. A compile-time error results if the required condition is not met and the error message displays the name of the symbol. A division by zero is clearly not allowed, and noting that a false value represents a zero value, the condition could be:

```cpp
1 / (sizeof(Target) <= sizeof(Source));
```

The interesting part is that this condition doesn’t result in any code at all. The enum’s value is a plain value that’s computed by the compiler while evaluating the expression:
23.2. TEMPLATE META PROGRAMMING

```cpp
enum {
    the_Target_size_exceeds_the_Source_size =
        1 / (sizeof(Target) <= sizeof(Source));
};
return reinterpret_cast<Target &>(source);
}
```

When `reinterpret_to_smaller_cast` is used to cast from `int` to `double` an error is produced by the compiler, like this:

```
error: enumerator value for 'the_Target_size_exceeds_the_Source_size'
        is not an integer constant
```

whereas no error is reported if, e.g., `reinterpret_to_smaller_cast<int>(doubleVar)` is requested with `doubleVar` defined as a `double`.

In the above example an `enum` was used to compute (at compile-time) a value that is illegal if an assumption is not met. The creative part is finding an appropriate expression.

Enum values are well suited for these situations as they do not consume any memory and their evaluation does not produce any executable code. They can be used to accumulate values too: the resulting enum value then contains a final value, computed by the compiler rather than by executable code as the next sections illustrate. In general, programs shouldn’t do run-time what they can do at compile-time and performing complex calculations resulting in constant values is a clear example of this principle.

### 23.2.1.1 Converting integral types to types

Another use of values buried inside templates is to ‘templatize’ simple scalar `int` values. This is useful in situations where a scalar value (often a `bool` value) is available to select a specialization but a type is required to base the selection on. This situation is shortly encountered (section 23.2.2).

Templatizing integral values is based on the fact that a class template together with its template arguments defines a type. E.g., `vector<int>` and `vector<double>` are different types.

Turning integral values into templates is easily done. Define a template (it does not have to have any contents at all) and store the integral value in an `enum`:

```cpp
template <int x>
struct IntType
{
    enum { value = x };  
};
```

As `IntType` does not have any members the ‘class IntType’ can be defined as ‘struct IntType’, saving us from having to type `public`:

Defining the `enum` value ‘value’ allows us to retrieve the value used at the instantiation at no cost in storage. Enum values are neither variables nor data members and thus have no address. They are mere values.

It’s easy to use the `struct IntType`. An anonymous or named object can be defined by specifying a value for its `int` non-type parameter. Example:
int main()
{
    IntType<1> it;
    cout << "IntType<1> objects have value: " << it.value << "\n" <<
        "IntType<2> objects are of a different type "
        "and have values " << IntType<2>().value << '\n';
}

Actually, neither the named object nor the anonymous object is required. As the enum is defined as a plain value, associated with the struct IntType we merely have to specify the specific int for which the struct IntType is defined to retrieve its 'value', like this:

int main()
{
    cout << "IntType<100>, no object, defines 'value': " << IntType<100>::value << "\n";
}

### 23.2.2 Selecting alternatives using templates

An essential characteristic of programming languages is that they allow the conditional execution of code. For this C++ offers the if and switch statements. If we want to be able to 'program the compiler' this feature must also be offered by templates.

Like templates storing values templates making choices do not require any code to be executed at run-time. The selection is purely made by the compiler, at compile-time. The essence of template meta programming is that we are not using or relying on any executable code. The result of a template meta program often is executable code, but that code is a function of decisions merely made by the compiler.

Template (member) functions are only instantiated when they are actually used. Consequently we can define specializations of functions that are mutually exclusive. Thus it is possible to define a specialization that can be compiled in situation one, but not in situation two and to define another specialization that can be compiled in situation two, but not in situation one. Using specializations code can be generated that is tailored to the demands of a particular situation.

A feature like this cannot be implemented in run-time executable code. For example, when designing a generic storage class the software engineer may intend to store value class type objects as well as objects of polymorphic class types in the final storage class. Thus the software engineer may conclude that the storage class should contain pointers to objects, rather than the objects themselves. The initial implementation attempt could look like this:

```cpp
template <typename Type>
void Storage::add(Type const &obj)
{
    d_data.push_back(
        d_ispolymorphic ?
            obj.clone()
        :
            new Type(obj)
    );
}
```
The intent is to use the clone member function of the class Type if Type is a polymorphic class and the standard copy constructor if Type is a value class.

Unfortunately, this scheme usually fails as value classes do not define clone member functions and polymorphic base classes should delete their copy constructors (cf. section 7.6). It doesn’t matter to the compiler that clone is never called for value classes and that the copy constructor is only available in value classes and not in polymorphic classes. It merely has some code to compile, and can’t do that because of missing members. It’s as simple as that.

23.2.2.1 Defining overloading members

Template meta programming comes to the rescue. Knowing that class template member functions are only instantiated when used, our plan is to design overloaded add member functions of which only one is going to be called (and thus instantiated). Our selection will be based on an additional (in addition to Type itself) template non-type parameter that indicates whether we’ll use Storage for polymorphic or non-polymorphic classes. Our class Storage starts like this:

```cpp
template <typename Type, bool isPolymorphic>
class Storage
```

Initially two overloaded versions of our add member are defined: one used with Storage objects storing polymorphic objects (using true as its template non-type argument) and one storing value class objects (using false as its template non-type argument).

Unfortunately we run into a small problem: functions cannot be overloaded by their argument values but only by their argument types. But the small problem may be solved. Realizing that types are defined by the combination of template names and their template arguments we may convert the values true and false into types using the knowledge from section 23.2.1.1 about how to convert integral values to types.

We’ll provide one (private) add member with a IntType<true> parameter (implementing the polymorphic class) and another (private) add member with a IntType<false> parameter (implementing the non-polymorphic class).

In addition to these two private members a third (public) member add is defined calling the appropriate private add member by providing an IntType argument, constructed from Storage’s template non-type parameter.

Here are the implementations of the three add members:

```cpp
// declared in Storage’s private section:

template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj, IntType<true>)
{
    d_data.push_back(obj.clone());
}

template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj, IntType<false>)
{
    d_data.push_back(new Type(obj));
}
```
// declared in Storage's public section:

template <typename Type, bool isPolymorphic>
void Storage<Type, isPolymorphic>::add(Type const &obj)
{
    add(obj, IntType<isPolymorphic>());
}

The appropriate add member is instantiated and called because a primitive value can be converted to a type. Each of the possible template non-type values is thus used to define an overloaded class template member function.

Since class template members are only instantiated when used only one of the overloaded private add members is instantiated. Since the other one is never called (and thus never instantiated) compilation errors are prevented.

### 23.2.2.2 Class structure as a function of template parameters

Some software engineers have reservations when thinking about the Storage class that uses pointers to store copies of value class objects. Their argument is that value class objects can very well be stored by value, rather than by pointer. They'd rather store value class objects by value and polymorphic class objects by pointer.

Such distinctions frequently occur in template meta programming and the following struct IfElse may be used to obtain one of two types, depending on a bool selector value.

First define the generic form of the template:

```cpp
template<bool selector, typename FirstType, typename SecondType>
struct IfElse
{
    typedef FirstType type;
};
```

Then define a partial specialization. The specialization represents a specific selector value (e.g., false) and leaves the remaining types open to further specification:

```cpp
template<typename FirstType, typename SecondType>
struct IfElse<false, FirstType, SecondType>
{
    typedef SecondType type;
};
```

The former (generic) definition associates FirstType with the IfElse::type type definition, the latter definition (partially specialized for the logical value false) associates SecondType with the IfElse::type type definition.

The IfElse template allows us to define class templates whose data organization is conditional to the template's parameters. Using IfElse the Storage class may define pointers to store copies of polymorphic class type objects and values to store value class type objects:

```cpp
template <typename Type, bool isPolymorphic>
```
class Storage {
    typedef typename IfElse<isPolymorphic, Type *, Type>::type DataType;

    std::vector<DataType> d_data;

    private:
        void add(Type const &obj, IntType<true>);
        void add(Type const &obj, IntType<false>);
    public:
        void add(Type const &obj);
    }

    template <typename Type, bool isPolymorphic>
    void Storage<Type, isPolymorphic>::add(Type const &obj, IntType<true>){
        d_data.push_back(obj.clone());
    }

    template <typename Type, bool isPolymorphic>
    void Storage<Type, isPolymorphic>::add(Type const &obj, IntType<false>){
        d_data.push_back(obj);
    }

    template <typename Type, bool isPolymorphic>
    void Storage<Type, isPolymorphic>::add(Type const &obj){
        add(obj, IntType<isPolymorphic>(),);
    }

    The above example uses IfElse’s type, defined by IfElse as either FirstType or SecondType. IfElse’s type defines the actual data type to use for Storage’s vector data type.

    The remarkable result in this example is that the data organization of the Storage class now depends on its template arguments. Since the isPolymorphic == true situation uses different data types than the isPolymorphic == false situation, the overloaded private add members can utilize this difference immediately. E.g., add(Type const &obj, IntType<false>) uses direct copy construction to store a copy of obj in d_vector.

    It is also possible to make a selection from multiple types as IfElse structs can be nested. Realize that using IfElse never has any effect on the size or execution time of the final executable program. The final program simply contains the appropriate type, conditional to the type that’s eventually selected.

23.2.2.3 An illustrative example

The next example, defining MapType as a map having plain types or pointers for either its key or value types, illustrates this approach:

    template <typename Key, typename Value, int selector>
    class Storage
CHAPTER 23. ADVANCED TEMPLATE USE

{
    typedef typename IfElse<
        selector == 1, // if selector == 1:
        map<Key, Value>, // use map<Key, Value>

    typename IfElse<
        selector == 2, // if selector == 2:
        map<Key, Value *>, // use map<Key, Value *>

    typename IfElse<
        selector == 3, // if selector == 3:
        map<Key *, Value>, // use map<Key *, Value>
        map<Key *, Value *> // use map<Key *, Value *>
    >::type
    >::type
    >::type
    MapType;

    MapType d_map;

    public:
        void add(Key const &key, Value const &value);
    private:
        void add(Key const &key, Value const &value, IntType<1>);
        ...
    };

template <typename Key, typename Value, int selector>
inline void Storage<selector, Key, Value>::add(Key const &key,
    Value const &value)
{
    add(key, value, IntType<selector>());
}

The principle used in the above examples is: if class templates may use data types that depend on
template non-type parameters, an IfElse struct can be used to select the appropriate data types.
Knowledge about the various data types may also be used to define overloaded member functions.
The implementations of these overloaded members may then be optimized to the various data types.
In programs only one of these alternate functions (the one that is optimized to the actually used
data types) will then be instantiated.

The private add functions define the same parameters as the public add wrapper function, but add
a specific IntType type, allowing the compiler to select the appropriate overloaded version based on
the template’s non-type selector parameter.

23.2.3 Templates: Iterations by Recursion

As there are no variables in template meta programming, there is no template equivalent to a for
or while statement. However, iterations can always be rewritten as recursions. Recursions are
supported by templates and so iterations can always be implemented as (tail) recursions.

To implement iterations by (tail) recursion do as follows:
• define a specialization implementing the end-condition;
• define all other steps using recursion.
• store intermediate values as enum values.

The compiler selects a more specialized template implementation over a more generic one. By the time the compiler reaches the end-condition the recursion stops since the specialization does not use recursion.

Most readers are probably familiar with the recursive implementation of the mathematical ‘factorial’ operator, commonly represented by the exclamation mark (!). Factorial n (so: n!) returns the successive products n * (n - 1) * (n - 2) * ... * 1, representing the number of ways n objects can be permuted. Interestingly, the factorial operator is itself usually defined by a recursive definition:

\[ n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{otherwise} \end{cases} \]

To compute n! from a template, a template Factorial can be defined using an int n template non-type parameter. A specialization is defined for the case n == 0. The generic implementation uses recursion according to the factorial definition. Furthermore, the Factorial template defines an enum value 'value' containing its factorial value. Here is the generic definition:

```cpp
template <int n>
struct Factorial
{
    enum { value = n * Factorial<n - 1>::value };
};
```

Note how the expression assigning a value to 'value' uses constant values that can be determined by the compiler. The value n is provided, and Factorial<n - 1> is computed using template meta programming. Factorial<n-1> in turn results in value that can be determined by the compiler (viz. Factorial<n-1>::value). Factorial<n-1>::value represents the value defined by the type Factorial<n - 1>. It is not the value returned by an object of that type. There are no objects here but merely values defined by types.

The recursion ends in a specialization. The compiler selects the specialization (provided for the terminating value 0) instead of the generic implementation whenever possible. Here is the specialization's implementation:

```cpp
template <>
struct Factorial<0>
{
    enum { value = 1 };
};
```

The Factorial template can be used to determine, compile time, the number of permutations of a fixed number of objects. E.g.,

```cpp
int main()
```
Once again, \texttt{Factorial<5>::value} is not evaluated at run-time, but at compile-time. The run-time equivalent of the above \texttt{cout} statement is, therefore:

```cpp
int main()
{
    cout << "The number of permutations of 5 objects = " << 120 << "\n";
}
```

### 23.3 User-defined literals

In addition to the literal operators discussed in section 11.12 the C++11 standard also offers a function template literal operator, matching the prototype

```cpp
template <char ...Chars>
Type operator "" _identifier() {
    return static_cast<size_t>( // forward Chars to NM2km
        NM2km<0, Chars ...>::value * 1.852);
}
```

The class template \texttt{NM2km} defines three non-type parameters: \texttt{acc} accumulates the value, \texttt{c} is the first character of the variadic non-type parameters, while \texttt{...Chars} represents the remaining non-type parameters, contained in a non-type parameter pack. Since \texttt{c} is, at each recursive call, the next character from the original non-type parameter pack, the value so far multiplied by 10 plus the
23.3. USER-DEFINED LITERALS

831

value of the next character is passed as the next accumulated value to its recursive call, together
with the remaining elements of the parameter pack, represented by Chars ...:
template <size_t acc, char c, char ...Chars>
struct NM2km
{
enum
{
value = NM2km<10 * acc + c - ’0’, Chars ...>::value
};
};
Eventually, the parameter pack is empty. For this case a partial specialization of NM2km is available:
template <size_t acc, char c>
// empty parameter pack
struct NM2km<acc, c>
{
enum
{
value = 10 * acc + c - ’0’
};
};
This works fine, but of course not in cases where binary, octal, or hexadecimal values must also be
interpreted. In that case we must first determine whether the first character(s) indicate a special
number system. This can be determined by the class template NM2kmBase, that is now called from
the _NM2km literal operator:
template <char ... Chars>
size_t constexpr operator "" _NM2km()
{
return static_cast<size_t>(
// forward Chars to NM2kmBase
NM2kmBase<Chars ...>::value * 1.852);
}
The NM2kmBase class template normally assumes the decimal number system, passing base value
10 and initial sum 0 to NM2km. The NM2km class template is provided with an additional (first)
non-type parameter representing the base value of the number system to use. Here is NM2kmBase:
template <char ...Chars>
struct NM2kmBase
{
enum
{
value = NM2km<10, 0, Chars ...>::value
};
};
Partial specializations handle the different number systems, by inspecting the first (one or two)
characters:
template <char ...Chars>


struct NM2kmBase<"0", Chars ...> // "0..."
{
    enum
    {
        value = NM2km<8, 0, Chars ...>::value
    };
};

template <char ...Chars>
struct NM2kmBase<"0", "b", Chars ...> // "0b...
{
    enum
    {
        value = NM2km<2, 0, Chars ...>::value
    };
};

template <char ...Chars>
struct NM2kmBase<"0", "x", Chars ...> // "0x...
{
    enum
    {
        value = NM2km<16, 0, Chars ...>::value
    };
};

NM2km is implemented as before, albeit that it can now handle various number systems. The conversion from character to numeric value is left to a small support function template, cVal:

template <char c>
int constexpr cVal()
{
    return '0' <= c <= '9' ? c - '0' : 10 + c - 'a';
}

template <size_t base, size_t acc, char c, char ...Chars>
struct NM2km
{
    enum
    {
        value = NM2km<base, base * acc + cVal<c>(),
                    Chars ...>::value
    };
};

template <size_t base, size_t acc, char c>
struct NM2km<base, acc, c>
{
    enum { value = base * acc + cVal<c>() }
};
23.4 Template template parameters

Consider the following situation: a software engineer is asked to design a storage class `Storage`. Data stored in `Storage` objects may either make and store copies of the data or store the data as received. `Storage` objects may also either use a vector or a linked list as its underlying storage medium. How should the engineer tackle this request? Should four different `Storage` classes be designed?

The engineer's first reaction could be to develop an all-in `Storage` class. It could have two data members, a list and a vector, and its constructor could be provided with maybe an enum value indicating whether the data itself or new copies should be stored. The enum value can be used to initialize a series of pointers to member functions performing the requested tasks (e.g., using a vector to store the data or a list to store copies).

Complex, but doable. Then the engineer is asked to modify the class: in the case of new copies a custom-made allocation scheme should be used rather than the standard `new` operator. He's also asked to allow the use of yet another type of container, in addition to the vector and the list that were already part of the design. Maybe a `deque` would be preferred or maybe even a `stack`.

It's clear that the approach aiming at implementing all functionality and all possible combinations in one class doesn't scale. The class `Storage` soon becomes a monolithic giant which is hard to understand, maintain, test, and deploy.

One of the reasons why the big, all-encompassing class is hard to deploy and understand is that a well-designed class should enforce constraints: the design of the class should, by itself, disallow certain operations, violations of which should be detected by the compiler, rather than by a program that might terminate in a fatal error.

Think about the above request. If the class offers both an interface to access the vector data storage and an interface to access the list data storage, then it's likely that the class offers an overloaded `operator[]` member to access elements in the vector. This member, however, will be syntactically present, but semantically invalid when the `list` data storage is selected, which doesn't support `operator[]`.

Sooner or later, users of the monolithic all-encompassing class `Storage` will fall into the trap of using `operator[]` even though they've selected the list as the underlying data storage. The compiler won't be able to detect the error, which only appears once the program is running, confusing its users.

The question remains: how should the engineer proceed, when confronted with the above questions? It's time to introduce policies.

23.4.1 Policy classes - I

A policy defines (in some contexts: prescribes) a particular kind of behavior. In C++ a policy class defines a certain part of the class interface. It may also define inner types, member functions, and data members.

In the previous section the problem of creating a class that might use any of a series of allocation schemes was introduced. These allocation schemes all depend on the actual data type to use, and so the 'template reflex' should kick in.

Allocation schemes should probably be defined as template classes, applying the appropriate allocation procedures to the data type at hand. When such allocation schemes are used by the familiar STL containers (like `std::vector`, `std::stack`, etc.), then such home-made allocation schemes
should probably be derived from `std::allocator`, to provide for the requirements made by these containers. The class template `std::allocator` is declared by the `<memory>` header file and the three allocation schemes developed here were all derived from `std::allocator`.

Using in-class implementations for brevity the following allocation classes could be defined:

- **No special allocation takes place, Data is used `as is`:**

  ```cpp
  template <typename Data>
  class PlainAlloc: public std::allocator<Data>
  {
    template<typename IData>
    friend std::ostream &operator<<(std::ostream &out, 
        PlainAlloc<IData> const &alloc);
    Data d_data;

    public:
      PlainAlloc() {}
      PlainAlloc(Data const &data) :
        d_data(data) {}
      PlainAlloc(PlainAlloc<Data> const &other) :
        d_data(other.d_data) {}
  };
  ```

- **The second allocation scheme uses the standard `new` operator to allocate a new copy of the data:**

  ```cpp
  template <typename Data>
  class NewAlloc: public std::allocator<Data>
  {
    template<typename IData>
    friend std::ostream &operator<<(std::ostream &out, 
        NewAlloc<IData> const &alloc);
    Data *d_data;

    public:
      NewAlloc() :
        d_data(0) {}
      NewAlloc(Data const &data) :
        d_data(new Data(data)) {}
      NewAlloc(NewAlloc<Data> const &other) :
        d_data(new Data(*other.d_data)) {}
      ~NewAlloc() {}
  };
  ```
The third allocation scheme uses the placement new operator (see section 9.1.5), requesting memory from a common pool (the implementation of the member request, obtaining the required amount of memory, is left as an exercise to the reader):

```cpp
template<typename Data>
class PlacementAlloc: public std::allocator<Data>
{
    template<typename IData>
    friend std::ostream &operator<< (std::ostream &out,
                                      PlacementAlloc<IData> const &alloc);

    Data *d_data;
    static char s_commonPool[];
    static char *s_free;

public:
    PlacementAlloc()
        : d_data(0)
    {}
    PlacementAlloc(Data const &data)
        : d_data(new(request()) Data(data))
    {}
    PlacementAlloc(PlacementAlloc<Data> const &other)
        : d_data(new(request()) Data(*other.d_data))
    {}
    ~PlacementAlloc()
    {
        d_data->~Data();
    }
    private:
        static char *request();
};
```

The above three classes define policies that may be selected by the user of the class Storage introduced in the previous section. In addition to these classes, additional allocation schemes could be implemented by the user as well.

To apply the proper allocation scheme to the class Storage, Storage should be designed as a class template itself. The class also needs a template type parameter allowing users to specify the data type.

The data type to be used by a particular allocation scheme could of course be specified when specifying the allocation scheme to use. The class Storage would then have two template type parameters, one for the data type, one for the allocation scheme:

```cpp
template<typename Data, typename Scheme>
class Storage ...
```
To use the class Storage we would then write, e.g.:

```cpp
Storage<string, NewAlloc<string>> storage;
```

Using Storage this way is fairly complex and potentially error-prone, as it requires the user to specify the data type twice. Instead, the allocation scheme should be specified using a new type of template parameter, not requiring the user to specify the data type required by the allocation scheme. This new kind of template parameter (in addition to the well-known template type parameter and template non-type parameter) is called the template template parameter.

### 23.4.2 Policy classes - II: template template parameters

Template template parameters allow us to specify a class template as a template parameter. By specifying a class template, it is possible to add a certain kind of behavior (called a policy) to an existing class template.

To specify an allocation policy, rather than an allocation type for the class Storage we rephrase its class template header: definition starts as follows:

```cpp
template <typename Data, template <typename> class Policy>
class Storage...
```

The second template parameter is new. It is a template template parameter. It has the following elements:

- The keyword `template`, starting the template template parameter;
- The keyword `template` is followed (between pointed brackets) by a list of template parameters that must be specified for the template template parameter. These parameters may be given names, but names are usually omitted as those names cannot be used in subsequent template definitions. On the other hand, providing formal names may help the reader of the template to understand the kind of templates that must be specified with the template template parameter.
- Template template parameters must match, in numbers and types (i.e., template type parameters, template non-type parameters, template template parameters) the template parameters that must be specified for the policy. This can be tricky, as some templates use default parameters that are hardly ever changed (like the allocation schemes for containers). Programmers may not immediately realize that these defaults exist and be confused when the compiler rejects such templates when trying to pass them as template template parameters for which these additional (default) parameters weren’t specified. However, the C++11 standard offers a solution for this problem in the form of template aliases, introduced in section 23.5.
- Following the bracketed list the keyword `class` must be specified. In this case, `typename` cannot be used.
- All parameters may be provided with default arguments. This is shown in the next example of a hypothetical class template:

```cpp
template <
    template <
        typename = std::string,
        int = 12,
        template <typename = int> class Inner = std::vector
```
Here, the class template `Demo` expects a template template parameter named `Policy`, expecting three template parameters: a template type parameter (by default `std::string`); a template non-type parameter (by default having value 12); and `Policy` itself expects a template template parameter, called `Inner`, by default using an `int` as its template type parameter.

Policy classes are often an integral part of the class under consideration. Because of this they are often deployed as base classes. In the example the class `Policy` could be used as a base class of the class `Storage`.

The policy operates on the class `Storage`'s data type. Therefore the policy is informed about that data type as well. Our class `Storage` now begins like this:

```cpp
template <typename Data, template <typename> class Policy>
class Storage: public Policy<Data>
```

This automatically allows us to use `Policy`'s members when implementing the members of the class `Storage`.

Our home-made allocation classes do not really provide us with many useful members. Except for the extraction operator they offer no immediate access to the data. This can easily be repaired by adding more members. E.g., the class `NewAlloc` could be augmented with operators allowing access to and modification of stored data:

```cpp
    operator Data &() // optionally add a 'const' member too
    {
        return *d_data;
    }
    NewAlloc &operator=(Data const &data)
    {
        *d_data = data;
    }
```

The other allocation classes could be given comparable members.

Let's use the allocation schemes in some real code. The next example shows how `Storage` can be defined using some data type and an allocation scheme. We start out again with a class `Storage`:

```cpp
template <typename Data, template <typename> class Allocate>
class Storage: public std::vector<Data, Allocate<Data>>
{};
```

That's all we have to do. Note that `std::vector` formally has two template parameters. The first one is the vector's data type, which is always specified; the second one is the allocator used by the vector. Usually the allocator is left unspecified (in which case the default STL allocator is used), but here it is mentioned explicitly, allowing us to pass our own allocation policy to `Storage`. 
All required functionality is inherited from the `vector` base class, while the policy is ‘factored into the equation’ using a template template parameter. Here’s an example showing how this is done:

```cpp
Storage<std::string, NewAlloc> storage;

copy(istream_iterator<std::string>(cin), istream_iterator<std::string>(),
     back_inserter(storage));

cout << "Element index 1 is " << storage[1] << 'n';
storage[1] = "hello";

copy(storage.begin(), storage.end(),
     ostream_iterator<NewAlloc<std::string> >(cout, "\n"));
```

Since `Storage` objects are also `std::vector` objects the STL `copy` function can be used in combination with the `back_inserter` iterator to add some data to the `storage` object. Its elements can be accessed and modified using the index operator. Then `NewAlloc<std::string>` objects are inserted into `cout` (also using the `copy` function).

Interestingly, this is not the end of the story. Remember that our intention was to create a class allowing us to specify the `storage type` as well. What if we don’t want to use a `vector`, but instead would like to use a `list`?

It’s easy to change `Storage`’s setup so that a completely different storage type can be used on request, like a `deque`. To implement this, the storage class is parameterized as well, using yet another template template parameter:

```cpp
template <typename Data, template <typename> class AllocationPolicy,
          template <typename, typename> class Container = std::vector>
class Storage: public Container<Data, AllocationPolicy<Data>>
{
};
```

The earlier example using a `Storage` object can be used again without requiring any modifications at all (except for the above redefinition). It clearly can’t be used with a `list` container, as the `list` lacks `operator[]`. Trying to do so is immediately recognized by the compiler, producing an error if an attempt is made to use `operator[]` on, e.g., a `list`. A `list` container, however can still be specified as the container to use. In that case a `Storage` is implemented as a `list`, offering `list`’s interface, rather than `vector`’s interface, to its users.

### 23.4.2.1 The destructor of Policy classes

In the previous section policy classes were used as base classes of template classes. This resulted in the interesting observation that a policy class may serve as a `base class` of a derived class. As a policy class may act as a base class, a pointer or reference to such a policy class can be used to point or refer to the derived class using the policy.

This situation, although legal, should be avoided for various reasons:

- Destruction of a derived class object using the base class’s destructor requires the implementation of a virtual destructor;

---

1. A complete example showing the definition of the allocation classes and the class `Storage` as well as its use is provided in the Annotation’s distribution in the file `yo/advancedtemplates/examples/storage.cc`. 

• A virtual destructor introduces overhead to a class that normally has no data members, but merely defines behavior: suddenly a vtable is required as well as a data member pointing to the vtable;

• Virtual member functions somewhat reduce the efficiency of code; virtual member functions use dynamic polymorphism, which in principle is undoing the advantages of static polymorphism as offered by templates;

• Virtual member functions in templates may result in code bloat: once an instantiation of a class’s member is required, the class’s vtable and all its virtual members must be implemented too.

To avoid these drawbacks, it is good practice to prevent the use of references or pointers to policy classes to refer or point to derived class objects. This is accomplished by providing policy classes with non-virtual protected destructors. With a non-virtual destructor there is no performance penalty and since its destructor is protected users cannot refer to classes derived from the policy class using a pointer or reference to the policy class.

23.4.3 Structure by Policy

Policy classes usually define behavior, not structure. Policy classes are normally used to parameterize some aspect of the behavior of classes that are derived from them. However, different policies may require different data members. These data members may also be defined by the policy classes. Policy classes may therefore be used to define both behavior and structure.

By providing a well-defined interface a class derived from a policy class may define member specializations using the different structures of policy classes to their advantage. For example, a plain pointer-based policy class could offer its functionality by resorting to C-style pointer juggling, whereas a vector-based policy class could use the vector’s members directly.

In this example a generic class template Size could be designed expecting a container-like policy using features commonly found in containers, defining the data (and hence the structure) of the container specified in the policy. E.g.:

```cpp
template <typename Data, template <typename> class Container>
struct Size: public Container<Data>
{
    size_t size()
    {
        // relies on the container’s ‘size()’
        // note: can’t use ‘this->size()’
        return Container<Data>::size();
    }
};
```

A specialization can now be defined for a much simpler storage class using, e.g., plain pointers (the implementation capitalizes on first and second, data members of std::pair. Cf. the example at the end of this section):

```cpp
template <typename Data>
struct Size<Data, Plain>: public Plain<Data>
{
    size_t size()
    {
        // relies on pointer data members
```
Depending on the intentions of the template’s author other members could be implemented as well.

To simplify the real use of the above templates, a generic wrapper class can be constructed: it uses the Size template matching the actually used storage type (e.g., a std::vector or some plain storage class) to define its structure:

```cpp
template <typename Data, template <typename> class Store>
class Wrapper: public Size<Data, Store>
{
};
```

The above classes could now be used as follows (en passant showing an extremely basic Plain class):

```cpp
#include <iostream>
#include <vector>

template <typename Data>
struct Plain: public std::pair<Data *, Data *>
{
};

int main()
{
    Wrapper<int, std::vector> wiv;
    std::cout << wiv.size() << "\n";

    Wrapper<int, Plain> wis;
    std::cout << wis.size() << "\n";
}
```

The wiv object now defines vector-data, the wis object merely defines a std::pair object’s data members.

### 23.5 Template aliases

In addition to function and class templates, C++11 also uses templates to define an alias for a set of types. This is called a template alias. Template aliases can be specialized. The name of a template alias is a type name.

Template aliases can be used as arguments to template template parameters. This allows us to avoid the ‘unexpected default parameters’ you may encounter when using template template parameters. E.g., defining a template specifying a template <typename> class Container is fine, but it is impossible to specify a container like vector or set as template template argument, as vector and set containers also define a second template parameter, specifying their allocation policy.

Template aliases are defined as using declarations, specifying an alias for an existing (maybe partially or fully specialized) template type. In the following example Vector is defined as an alias for vector:

```cpp
template <typename Type>
using Vector = std::vector<Type>;

Vector<int> vi; // same as std::vector<int>
std::vector<int> vi2(vi); // copy construction: OK

So, what’s the point of doing this? Looking at the vector container, we see that it defines two, rather than one, template parameters, the second parameter being the allocation policy _Alloc, by default set to std::allocator<_Tp>:

```
template<typename _Tp, typename _Alloc = std::allocator<_Tp> >
class vector: ...
```

Now define a class template Generic defining a template template parameter:

```
template <typename Type, template <typename> class Container>
class Generic: public Container<Type> {
  ...
};
```

Most likely, Generic offers members made available by the container that is actually used to create the Generic object, and adds to those some members of its own. However, a simple container like std::vector cannot be used, as std::vector doesn’t match a template <typename> class Container> parameter; it requires a template <typename, typename> class Container> template template parameter.

The Vector template alias, however, is defined as a template having one template type parameter, and it uses the vector’s default allocator. Consequently, passing a Vector to Generic works fine:

```
Generic<int, Vector> giv; // OK
Generic<int, std::vector> err; // won’t compile: 2nd argument mismatch
```

With the aid of a small template alias it is also possible to use a completely different kind of container, like a map, with Generic:

```
template <typename Type>
using MapString = std::map<Type, std::string>;

Generic<int, MapString> gim; // uses map<int, string>
```

### 23.6 Trait classes

Scattered over the std namespace trait classes are found. E.g., most C++ programmers have seen the compiler mentioning ‘std::char_traits<char>’ when performing an illegal operation on std::string objects, as in std::string s(1).

Trait classes are used to make compile-time decisions about types. Trait classes allow us to apply the proper code to the proper data type, be it a pointer, a reference, or a plain value, all this maybe in combination with const. The particular type of data to use can be inferred from the actual type that is specified (or implied) when the template is used. This can be fully automated, not requiring the template writer to make any decision.
Trait classes allow us to develop a template \(\langle\text{typename Type1, typename Type2, ...}\rangle\) without the need to specify many specializations covering all combinations of, e.g., values, (const) pointers, or (const) references, which would soon result in an unmaintainable exponential explosion of template specializations (e.g., allowing these five different types for each template parameter already results in 25 combinations when two template type parameters are used: each must be covered by potentially different specializations).

Having available a trait class, the actual type can be inferred compile time, allowing the compiler to deduce whether or not the actual type is a pointer, a pointer to a member, or a const pointer, and to make comparable deductions in case the actual type is, e.g., an lvalue or rvalue reference type. This in turn allows us to write templates that define types like `argument_type`, `first_argument_type`, `second_argument_type` and `result_type`, which are required by several generic algorithms (e.g., `count_if()`).

A trait class usually performs no behavior. I.e., it has no constructor and no members that can be called. Instead, it defines a series of types and `enum` values that have certain values depending on the actual type that is passed to the trait class template. The compiler uses one of a set of available specializations to select the one appropriate for an actual template type parameter.

The point of departure when defining a trait template is a plain vanilla `struct`, defining the characteristics of a plain value type like an `int`. This sets the stage for specific specializations, modifying the characteristics for any other type that could be specified for the template.

To make matters concrete, assume the intent is to create a trait class `BasicTraits` telling us whether a type is a plain value type, a pointer type, an lvalue reference type or an rvalue reference type (all of which may or may not be `const` types).

Whatever the actually provided type, we want to be able to determine the ‘plain’ type (i.e., the type without any modifiers, pointers or references), the ‘pointer type’ and the ‘reference type’, allowing us to define in all cases, e.g., an rvalue reference to its built-in type, even though we passed a const pointer to that type.

Our point of departure, as mentioned, is a plain `struct` defining the required parameter. Maybe something like this:

```cpp
template <typename T>
struct Basic
{
    typedef T Type;
    enum
    {
        isPointer = false,
        isConst = false,
        isRef = false,
        isRRef = false
    };
};
```

Although some conclusions can be drawn by combining various `enum` values (e.g., a plain type is not a pointer or a reference or a rvalue reference or a const), it is good practice to provide a full implementation of trait classes, not requiring its users to construct these logical expressions themselves. Therefore, the basic decisions in a trait class are usually made by a nested trait class, leaving the task of creating appropriate logical expressions to a surrounding trait class.

So, the `struct Basic` defines the generic form of our inner trait class. Specializations handle specific details. E.g., a pointer type is recognized by the following specialization:

```cpp
```
template<typename T>
struct Basic<T*> {
    typedef T Type;
    enum {
        isPointer = true,
        isConst = false,
        isRef = false,
        isRRef = false
    };
};

whereas a pointer to a const type is matched with the next specialization:

template<typename T>
struct Basic<T const*> {
    typedef T Type;
    enum {
        isPointer = true,
        isConst = true,
        isRef = false,
        isRRef = false
    };
};

Several other specializations should be defined: e.g., recognizing const value types or (rvalue) reference types. Eventually all these specializations are implemented as nested structs of an outer class BasicTraits, offering the public traits class interface. The outline of the outer trait class is:

template<typename TypeParam>
class BasicTraits {
    // Define specializations of the template 'Base' here

    public:
        BasicTraits(BasicTraits const &other) = delete;

        typedef typename Basic<TypeParam>::Type ValueType;
        typedef ValueType*PtrType;
        typedef ValueType&RefType;
        typedef ValueType&&RvalueRefType;

        enum {
            isPointerType = Basic<TypeParam>::isPointer,
            isReferenceType = Basic<TypeParam>::isRef,
            isRvalueReferenceType = Basic<TypeParam>::isRRef,
            isConst = Basic<TypeParam>::isConst,
            isPlainType = not (isPointerType or isReferenceType or
                               isRvalueReferenceType or isConst)
The trait class’s public interface explicitly deletes its copy constructor. As it therefore defines no constructor at all and as it has no static members it does not offer any run-time executable code. All the trait class’s facilities must therefore be used compile time.

A trait class template can be used to obtain the proper type, irrespective of the template type argument provided. It can also be used to select a proper specialization that depends on, e.g., the const-ness of a template type. Example:

```cpp
BasicTraits<int>::ValueType value = 12;
BasicTraits<int const *>::RvalueRefType rvalue = int(10);
BasicTraits<int const &&>::PtrType ptr = new int(14);
cout << value << ' ' << rvalue << ' ' << *ptr << '
';
```

### 23.6.1 Distinguishing class from non-class types

In the previous section the `TypeTrait` trait class was developed. Using specialized versions of a nested `struct` `Type` modifiers, pointers, references and values could be distinguished.

Knowing whether a type is a class type or not (e.g., the type represents a primitive type) could also be a useful bit of knowledge to a template developer. The class template developer might want to define a specialization when the template’s type parameter represents a class type (maybe using some member function that should be available) and another specialization for non-class types.

This section addresses the question how a trait class can distinguish class types from non-class types.

In order to distinguish classes from non-class types a distinguishing feature that can be used at compile-time must be found. It may take some thinking to find such a distinguishing characteristic, but a good candidate eventually is found in the pointer to members syntactic construct. Pointers to
members are only available for classes. Using the pointer to member construct as the distinguishing characteristic, a specialization can be developed that uses the pointer to member if available. Another specialization (or the generic template) does something else if the pointer to member construction is not available.

How can we distinguish a pointer to a member from a 'generic situation', not being a pointer to a member? Fortunately, such a distinction is possible. A function template specialization can be defined having a parameter which is a pointer to a member function. The generic function template then accepts any other argument. The compiler selects the former (specialized) function when the provided type is a class type as class types may support a pointer to a member. The interesting verb here is 'may': the class does not have to define a pointer to member.

Furthermore, the compiler does not actually call any function: we're talking compile-time here. All the compiler does is to select the appropriate function by evaluating a constant expression.

So, our intended function template now looks like this:

```cpp
template <typename ClassType>
static 'some returntype' fun(void (ClassType::*)());
```

The function's return type ('(some returntype)') will be defined shortly. Let's first have a closer look at the function's parameter. The function's parameter defines a pointer to a member returning void. Such a function does not have to exist for the concrete class-type that's specified when the function is used. In fact, no implementation is provided. The function `fun` is only declared as a static member in the trait class. It's not implemented and no trait class object is required to call it. What, then, is its use?

To answer the question we now have a look at the generic function template that should be used when the template's argument is not a class type. The language offers a 'worst case' parameter in its ellipsis parameter list. The ellipsis is a final resort the compiler may turn to if everything else fails. The generic function template specifies a plain ellipsis in its parameter list:

```cpp
template <typename NonClassType>
static 'some returntype' fun(...);
```

It would be an error to define the generic alternative as a function expecting an int. The compiler, when confronted with alternatives, favors the simplest, most specified alternative over a more complex, generic one. So, when providing `fun` with an argument it selects int whenever possible and it won't select `fun(void (ClassType::*)());`. When given the choice between `fun(void (ClassType::*)())` and `fun(...)` it selects the former unless it can't do that.

The question now becomes: what argument can be used for both a pointer to a member and for the ellipsis? Actually, there is such a 'one size fits all' argument: 0. The value 0 can be passed as argument value to functions defining an ellipsis parameter and to functions defining a pointers-to-member parameter.

But 0 does not specify a particular class. Therefore, `fun` must specify an explicit template argument, appearing in our code as `fun<Type>(0)`, with `Type` being the template type parameter of the trait class.

Now for the return type. The function's return type cannot be a simple value (like true or false). Our eventual intent is to provide the trait class with an enum telling us whether the trait class's template argument represents a class type or not. That enum becomes something like this:

```cpp
enum { isClass = some class/non-class distinguishing expression };
```
The distinguishing expression cannot be
\[
\text{enum } \{ \text{isClass} = \text{fun<Type>(0)} \} ;
\]
as $\text{fun<Type>(0)}$ is not a constant expression and enum values \textit{must} be defined by constant expressions so they can be determined at compile-time.

To determine $\text{isClass}$'s value we must find an expression allowing for compile-time discriminations between $\text{fun<Type>(...)}$ and $\text{fun<Type>(void (Type::*)(void))}$.

In situations like these the $\text{sizeof}$ operator often is our tool of choice as it is evaluated at compile-time. By defining different sized return types for the two $\text{fun}$ declarations we are able to distinguish (at compile-time) which of the two $\text{fun}$ alternatives is selected by the compiler.

The $\text{char}$ type is by definition a type having size 1. By defining another type containing two consecutive $\text{char}$ values a larger type is obtained. A $\text{char[2]}$ is of course not a type, but a $\text{char[2]}$ can be defined as a data member of a struct, and a struct $\text{does}$ define a type. That struct then has a size exceeding 1. E.g.,

\begin{verbatim}
struct Char2
{
    char data[2];
};
\end{verbatim}

$\text{Char2}$ can be defined as a nested type of our traits class. The two $\text{fun}$ function template declarations become:

\begin{verbatim}
template <typename ClassType>
static Char2 fun(void (ClassType::*)(void));

template <typename NonClassType>
static char fun(...);
\end{verbatim}

Since $\text{sizeof}$ expressions are evaluated at compile-time we can now determine $\text{isClass}$'s value:

\[
\text{enum } \{ \text{isClass} = \text{sizeof(fun<Type>(0)) == sizeof(Char2)} \} ;
\]

This expression has several interesting implications:

- no $\text{fun}$ function template is ever instantiated;
- the compiler considers $\text{Type}$ and selects $\text{fun}$'s function template specialization if $\text{Type}$ is a class type and the generic function template if not;
- From the selected function it determines the return type and thus the return type's size;
- Resulting in the proper evaluation of $\text{isClass}$.

Without requiring any instantiation the trait class can now provide an answer to the question whether a template type argument represents a class type or not. Marvelous!
23.6. TRAIT CLASSES

23.6.2 Available type traits

The C++11 standard offers many facilities to identify and modify characteristics of types. Before using these facilities the `<type_traits>` header file must be included.

All facilities offered by `type_traits` are defined in the `std` namespace (omitted from the examples given below), allowing programmers to determine various characteristics of types and values.

In the description of available type traits the following concepts are encountered:

- **arithmetic type**: any integral or floating point type;
- **class type**: not a union type, without non-static data members, without virtual members, without virtual or non-empty base classes;
- **compound type**:
  - arrays of objects of a given type;
  - functions, which have parameters of a given type returning `void` or objects;
  - pointers to `void`, to objects, to functions, or to non-static class members;
  - references to objects or functions;
  - class, union or enumeration types;
- **fundamental type**: a built-in type;
- **integral type**: all types whose values represent integral numbers, as well as `bool` and all built-in types representing (possibly unicode) characters;
- **literal type**: a literal type is a scalar type; a trivial class type; or an array of literal type elements;
- **is_nothrow_... type trait**: a type trait to determine whether its template argument supports the specified non-throwing member. Such type traits return `false` unless `noexcept(true)` is used at the function’s declaration. E.g.,

```cpp
struct NoThrow
{
    NoThrow &operator=(SomeType const &rhs) noexcept(true);
};
```

- **trivial type**: trivial types are scalar types, trivial class types, arrays of such types and cv-qualified versions of these types;
- **trivial class type**: a class type having a trivial copy constructor, no non-trivial move constructor, a trivial destructor, a trivial default constructor or at least one `constexpr` constructor other than the copy or move constructor, and only has non-static data members and base classes of literal types;
- **trivial member function**: trivial member functions are never declared (other than `default`) in their class interfaces and (for default constructors or assignment operators) only perform byte-by-byte actions. Here are two examples: `struct Pod` only has trivial members as it doesn’t explicitly declare any member function and its data member is *plain old data*. `struct Nonpod` is *not* plain old data. Although it doesn’t explicitly declare any member function either, its data member is a `std::string`, which itself isn’t plain old data as `std::string` has non-trivial constructors:

```cpp
struct Pod
```
When *type-condition* applies to a type, it must be a complete type, `void`, or an array of unknown size;

The following type traits are provided:

- `add_const<typename Type>::type` to add `const` to `Type`;
- `add_cv<typename Type>::type` to add `const volatile` to `Type`;
- `add_lvalue_reference<typename Type>::type` to add a lvalue reference to `Type`;
- `add_rvalue_reference<typename Type>::type` to add an rvalue reference to `Type`;
- `add_pointer<typename Type>::type` to add a pointer to `Type`;
- `add_volatile<typename Type>::type` to add `volatile` to `Type`;
- `conditional<bool cond, typename TrueType, typename FalseType>::type` to conditionally use `TrueType` if `cond` is true, `FalseType` if not;
- `enable_if<bool cond, typename Type>::type` to conditionally define `Type` if `cond` is true;
- `has_nothrow_assign<typename Type>::value` to determine whether `Type` has an assignment operator not throwing exceptions;
- `has_nothrow_copy_constructor<typename Type>::value` to determine whether `Type` has a copy constructor not throwing exceptions;
- `has_nothrow_default_constructor<typename Type>::value` to determine whether `Type` has a constructor not throwing exceptions;
- `has_nothrow_destructor<typename Type>::value` to determine whether `Type` has a destructor not throwing exceptions;
- `has_trivial_assign<typename Type>::value` to determine whether `Type` has a trivial assignment operator;
- `has_trivial_copy_constructor<typename Type>::value` to determine whether `Type` has a trivial copy constructor;
- `has_trivial_default_constructor<typename Type>::value` to determine whether `Type` has a trivial default constructor;
- `has_trivial_destructor<typename Type>::value` to determine whether `Type` has a trivial destructor;
- `has_virtual_destructor<typename Type>::value` to determine whether `Type` has a virtual destructor;
• `is_abstract<typename Type>::value` to determine whether `Type` is an abstract type (e.g., an abstract base class) (*type-condition* applies);

• `is_arithmetic<typename Type>::value` to determine whether `Type` is an arithmetic type;

• `is_array<typename Type>::value` to determine whether `Type` is an array type;

• `is_assignable<typename To, typename From>::value` to determine whether an object of type `From` can be assigned to an object of type `To` (*type-condition* applies);

• `is_base_of<typename Base, typename Derived>::value` to determine whether `Base` is a base class of type `Derived`;

• `is_class<typename Type>::value` to determine whether `Type` is a class type;

• `is_compound<typename Type>::value` to determine whether `Type` is a compound type;

• `is_const<typename Type>::value` to determine whether `Type` is a const type;

• `is_constructible<typename Type, typename ...Args>::value` to determine whether `Type` is constructible from arguments in the `Args` parameter pack (*type-condition* applies to all types in `Args`);

• `is_convertible<typename From, typename To>::value` to determine whether a type `From` may be converted to a type `To` using a `static_cast`;

• `is_copy_assignable<typename Type>::value` to determine whether `Type` supports copy assignment (*type-condition* applies);

• `is_copy_constructible<typename Type>::value` to determine whether `Type` supports copy construction (*type-condition* applies);

• `is_default_constructible<typename Type>::value` to determine whether `Type` supports a default constructor (*type-condition* applies);

• `is_destructible<typename Type>::value` to determine whether `Type` has a non-deleted destructor (*type-condition* applies);

• `is_empty<typename Type>::value` to determine whether `Type` is a class type (not a union type), without non-static data members, virtual members, virtual or non-empty base classes (*type-condition* applies);

• `is_enum<typename Type>::value` to determine whether `Type` is an enum type;

• `is_floating_point<typename Type>::value` to determine whether `Type` is a floating point type;

• `is_function<typename Type>::value` to determine whether `Type` is a function type;

• `is_fundamental<typename Type>::value` to determine whether `Type` is a fundamental type;

• `is_integral<typename Type>::value` to determine whether `Type` is an integral type;

• `is_literal_type<typename Type>::value` to determine whether `Type` is a literal type (*type-condition* applies);

• `is_lvalue_reference<typename Type>::value` to determine whether `Type` is an lvalue reference;
• `is_member_function_pointer<typename Type>::value` to determine whether `Type` is a pointer to a non-static member function;

• `is_member_object_pointer<typename Type>::value` to determine whether `Type` is a pointer to a non-static data member;

• `is_member_pointer<typename Type>::value` to determine whether `Type` is a pointer to a member function;

• `is_move_assignable<typename Type>::value` to determine whether `Type` supports move assignment (`type-condition` applies);

• `is_move_constructible<typename Type>::value` to determine whether `Type` supports move construction (`type-condition` applies);

• `is_nothrow_assignable<typename To, typename From>::value` to determine whether `Type` supports an assignment operator not throwing exceptions (`type-condition` applies);

• `is_nothrow_constructible<typename Type, typename ...Args>::value` to determine whether a `Type` object can be constructed from arguments of types mentioned in the parameter pack not throwing exceptions (`type-condition` applies);

• `is_nothrow_copy_assignable<typename Type>::value` to determine whether `Type` supports a copy-assignment operator not throwing exceptions (`type-condition` applies);

• `is_nothrow_copy_constructible<typename Type>::value` to determine whether `Type` supports copy construction not throwing exceptions (`type-condition` applies);

• `is_nothrow_default_constructible<typename Type>::value` to determine whether `Type` supports a default constructor not throwing exceptions (`type-condition` applies);

• `is_nothrow_destructible<typename Type>::value` to determine whether `Type` supports a destructor not throwing exceptions (`type-condition` applies);

• `is_nothrow_move_assignable<typename Type>::value` to determine whether `Type` supports move assignment not throwing exceptions (`type-condition` applies);

• `is_nothrow_move_constructible<typename Type>::value` to determine whether `Type` supports a move constructor not throwing exceptions (`type-condition` applies);

• `is_object<typename Type>::value` to determine whether `Type` is an object (in contrast to scalar) type;

• `is_pod<typename Type>::value` to determine whether `Type` is `plain old data` (`type-condition` applies);

• `is_pointer<typename Type>::value` to determine whether `Type` is a pointer type;

• `is_polymorphic<typename Type>::value` to determine whether `Type` is a polymorphic type (`type-condition` applies);

• `is_reference<typename Type>::value` to determine whether `Type` is an (lvalue or rvalue) reference;

• `is_rvalue_reference<typename Type>::value` to determine whether `Type` is an rvalue reference;

• `is_same<typename First, typename Second>::value` to determine whether types `First` and `Second` are identical;
• `is_scalar<typename Type>::value` to determine whether `Type` is a scalar type (in contrast to an object type);

• `is_signed<typename Type>::value` to determine whether `Type` is a signed type;

• `is_standard_layout<typename Type>::value` to determine whether `Type` offers the standard layout (`type-condition` applies);

• `is_trivial<typename Type>::value` to determine whether `Type` is a trivial type (`type-condition` applies);

• `is_trivially_assignable<typename Dest, typename Src>::value` to determine whether an object or value of type `Src` can trivially be assigned to an object of type `Dest` (`type-condition` applies);

• `is_trivially_constructible<typename Type, typename ...Args>::value` to determine whether `Type` is trivially constructible from arguments in the `Args` parameter pack (`type-condition` applies to all types in `Args`);

• `is_trivially_copy_assignable<typename Type>::value` to determine whether `Type` supports a trivial assignment operator (`type-condition` applies);

• `is_trivially_copy_constructible<typename Type>::value` to determine whether `Type` is trivially copy-constructible (`type-condition` applies);

• `is_trivially_copyable<typename Type>::value` to determine whether `Type` is trivially copyable (`type-condition` applies);

• `is_trivially_default_constructible<typename Type>::value` to determine whether `Type` supports a trivial default constructor (`type-condition` applies);

• `is_trivially_default_destructible<typename Type>::value` to determine whether `Type` supports a trivial default destructor (`type-condition` applies);

• `is_trivially_move_assignable<typename Type>::value` to determine whether `Type` supports a trivial assignment operator (`type-condition` applies);

• `is_trivially_move_constructible<typename Type>::value` to determine whether `Type` is trivially move-constructible (`type-condition` applies);

• `is_union<typename Type>::value` to determine whether `Type` is a union type;

• `is_unsigned<typename Type>::value` to determine whether `Type` is an unsigned type;

• `is_void<typename Type>::value` to determine whether `Type` is void;

• `is_volatile<typename Type>::value` to determine whether `Type` is a volatile qualified type;

• `make_signed<typename Type>::type` to construct a signed type;

• `make_unsigned<typename Type>::type` to construct an unsigned type;

• `remove_all_extents<typename Type>::type` if `Type` is a (possibly multidimensional) array of `ElementType` values or objects then typedef type equals `ElementType`;

• `remove_const<typename Type>::type` to remove `const` from `Type`;

• `remove_cv<typename Type>::type` to remove `const` and/or `volatile` from `Type`;

• `remove_extent<typename Type>::type` if `Type` is an array of `ElementType` values or objects then typedef type equals `ElementType`. With multi-dimensional arrays `ElementType` is the type of the array from which its first array dimension has been removed;
• remove_pointer<typename Type>::type to remove a pointer from Type;
• remove_reference<typename Type>::type to remove a reference from Type;
• remove_volatile<typename Type>::type to remove volatile from Type;

23.7 Using ‘noexcept’ when offering the ‘strong guarantee’

When throwing exceptions while trying to achieve the strong guarantee a function’s actions are usually split in two parts

• First, usually on a temporary object, all operations that may throw are performed (which doesn’t affect the target object)
• Then, the target object is modified using operations that offer the noexcept guarantee.

The actions in the first step might be made move aware by using std::move (e.g., to assign the source’s values to a (possibly temporary) destination). However, using std::move can easily affect the source (e.g., when extending the source’s memory, moving the existing data to its new locations), which breaks the first step’s assumption, as the target object is now modified.

In this case (and generally) the move operation should not be allowed to throw exceptions. This, in turn, implies that it is difficult to write code which must offer a non-throwing moving constructor, if it uses (external) data types over which the moving constructor has no control. E.g.,

```cpp
template <typename Type>
class MyClass
{
    Type d_f;
public:
    MyClass() = default;
    MyClass(MyClass &&tmp)
    :
        d_f(move(tmp.d_f))
    {}
};
```

Here, MyClass's author has no control over the design of Type. If Foreign merely has a (possibly throwing) copy constructor, then the following code breaks the no-throw assumption underlying move constructors:

```cpp
MyClass<Foreign> s2(move(MyClass<Foreign>()));
```

If templates are able to detect whether Type has non-throwing move constructors then their implementations may be optimized by calling these move constructors (already modifying their targets in the first part of code offering the strong guarantee) in situations where otherwise the non-modifying, but more expensive copy constructor has to be used.

The noexcept keyword was introduced to allow such templates to perform such optimizations. As with throw lists, checking for noexcept is a run-time check, but the consequence of violating a noexcept declaration are more serious than violating a throw list: violating noexcept results in calling std::terminate, terminating the program, possibly without unwinding the stack. In the
23.7. USING 'NOEXCEPT' WHEN OFFERING THE 'STRONG GUARANTEE'

context of the previous example, the following code is flawlessly accepted by the compiler, demonstrating that there is no compile-time checking of noexcept:

```cpp
class Foreign
{
public:
    Foreign() = default;
    Foreign(Foreign const &other) noexcept
    {
        throw 1;
    }
};
```

However, when this class's copy constructor is called, execution aborts with the following message:

```
terminate called after throwing an instance of 'int'
Abort
```

Keep in mind that the current purpose of noexcept is to allow templates to optimize their code by using move operations where the code must also be able to offer the strong exception guarantee. Since noexcept also offers the conditional noexcept (condition) syntax (with noexcept(true) and noexcept having identical semantics), noexcept can be made conditional to the 'noexcepting' nature of template types. Note that this is not possible with throw lists.

The following rules of thumb by be used to decide whether or not to use noexcept in your code:

- **General rule:** don’t use noexcept (this is identical to the advise given for throw lists);

- Default implementations of constructors, copy- and move-assignment operators and destructors are provided with noexcept(true) if the compiler can deduct that composing types also offer noexcept(true), allowing template optimizations using move operations where possible.

- Functions provided with noexcept declarations may still throw exceptions (see the example given above). In the end noexcept merely means that if such a function throws an exception std::terminate rather than std::unexpected is called.

- Functions previously provided with an empty throw list (throw()) should now be provided with noexcept(true).

- noexcept specifications are required when using the following std traits (declared in the <type_traits> header file):
  - is_nothrow_constructible
  - is_nothrow_default_constructible
  - is_nothrow_move_constructible
  - is_nothrow_copy_constructible
  - is_nothrowAssignable
  - is_nothrow_move_assignable
  - is_nothrow_copy_assignable
  - is_nothrow_destructible
These type traits provide the member constant \texttt{value} which is \texttt{true} if the class (and possibly its argument type list) matches the characteristic after which the trait was named. E.g., if \texttt{MyClass(string const \&) noexcept} is a constructor, then

\begin{verbatim}
std::is_nothrow_constructible<MyClass, string>::value
\end{verbatim}
equals \texttt{true}. For the named members (like \texttt{is_nothrow_move_constructible}) parameter types do not have to be specified, as they are implied. E.g.,

\begin{verbatim}
std::is_nothrow_move_constructible<MyClass>::value
\end{verbatim}
returns \texttt{true} if the move constructor has the \texttt{noexcept} modifier.

### 23.8 More conversions to class types

#### 23.8.1 Types to types

Although \texttt{class} templates may be partially specialized, \texttt{function} templates may not. At times this is annoying. Assume a function template is available implementing a certain unary operator that could be used with the \texttt{transform} generic algorithm (cf. section 19.1.63):

\begin{verbatim}
template <typename Return, typename Argument>
Return chop(Argument const &arg)
{
    return Return(arg);
}
\end{verbatim}

Furthermore assume that if \texttt{Return} is \texttt{std::string} then the above implementation should not be used. Instead, with \texttt{std::string} a second argument \texttt{1} should always be provided. If \texttt{Argument} is a \texttt{C++} string, this would allow us to, e.g., return a copy of \texttt{arg} from which its first character has been chopped off.

Since \texttt{chop} is a function, it is not possible to define a partial specialization like this:

\begin{verbatim}
template <typename Argument>    // This won’t compile!
std::string chop<std::string, Argument>(Argument const &arg)
{
    return string(arg, 1);
}
\end{verbatim}

Although a function template cannot be partially specialized it \texttt{is} possible to use overloading, defining a second, dummy, \texttt{string} parameter:

\begin{verbatim}
template <typename Argument>
std::string chop(Argument const &arg, std::string)
{
    return string(arg, 1);
}
\end{verbatim}

Now it \texttt{is} possible to distinguish the two cases, but at the expense of a more complex function call. Moreover, in code this function may require the use of the \texttt{bind2nd} binder (cf. section 18.1.4) to
provide the dummy second argument, or it may require a (possibly expensive to construct) dummy argument to allow the compiler to choose among the two overloaded function templates.

Instead of providing a `string` dummy argument the functions could use the `IntType` template (cf. section 23.2.1.1) to select the proper overloaded version. E.g., `IntType<0>` could be defined as the type of the second argument of the first overloaded `chop` function, and `IntType<1>` could be used for the second overloaded function. From the point of view of program efficiency this is an attractive option, as the provided `IntType` objects are extremely lightweight. `IntType` objects contain no data at all. But there’s also an obvious disadvantage as there is no intuitively clear association between the `int` value used and the intended type.

Instead of defining arbitrary `IntType` types it is more attractive to use another lightweight solution, using an automatic type-to-type association. The `struct TypeType` is a lightweight type wrapper, much like `IntType`. Here is its definition:

```cpp
template <typename T>
struct TypeType
{
    typedef T Type;
};
```

`TypeType` is also a lightweight type as it doesn’t have any data fields either. `TypeType` allows us to use a natural type association for `chop`’s second argument. E.g, the overloaded functions can now be defined as follows:

```cpp
template <typename Return, typename Argument>
Return chop(Argument const &arg, TypeType<Argument> )
{
    return Return(arg);
}

template <typename Argument>
std::string chop(Argument const &arg, TypeType<std::string> )
{
    return std::string(arg, 1);
}
```

Using the above implementations any type can be specified for `Result`. If it happens to be a `std::string` the appropriate overloaded version is automatically selected. The following additional overload of the function `chop` capitalizes on this:

```cpp
template <typename Result>
Result chop(char const *txt) // char const * could also be a 2nd
    // template type parameter
{
    return chop(std::string(txt), TypeType<Result>());
}
```

Using the third `chop` function, the following statement produces the text ‘ello world’:

```cpp
cout << chop<string>("hello world") << ‘\n’;
```

Template functions do not support partial specializations. But they can be overloaded. By providing overloads with dummy type-arguments that depend on other parameters and calling these overloads
from a overloaded function that does not require the dummy type argument a situation similar to partial specializations with class templates can often be realized.

23.8.2 An empty type

At times (cf. section 23.9) an empty struct is a useful tool. It can be used as a type acting analogously to the final 0-byte in NTB-strings. It can simply be defined as:

```cpp
struct NullType
{
};
```

23.8.3 Type convertibility

In what situations can a type \( T \) be used as a ‘stand in’ for another type \( U \)? Since C++ is a strongly typed language the answer is surprisingly simple: \( T \)s can be used instead of \( U \)s if a \( T \) is accepted as argument in cases where \( U \)s are requested.

This reasoning is behind the following class which can be used to determine whether a type \( T \) can be used where a type \( U \) is expected. The interesting part is that no code is actually generated or executed. All decisions are made by the compiler.

In the second part of this section we’ll show how the code developed in the first part can be used to detect whether a class \( B \) is a base class of another class \( D \) (the is_base_of template (cf. section 23.6.2) also provides an answer to this question). The code developed here closely follows the example provided by Alexandrescu (2001, p. 35).

First, a function `test` is designed accepting a type \( U \). The function `test` returns a value of the as yet unknown type `Convertible`:

```cpp
Convertible test(U const &);
```

The function `test` is never implemented. It is only declared. If a type \( T \) can be used instead of a type \( U \) then \( T \) can also be passed as argument to the above `test` function.

On the other hand, if the alternate type \( T \) cannot be used where a \( U \) is expected, then the compiler won’t be able to use the above `test` function. Instead, it uses an alternative function that has a lower selection priority but that can always be used with any \( T \) type.

C (and C++) offer a very general parameter list, a parameter list that is always considered acceptable. This parameter list is the familiar ellipsis which represents the worst case the compiler may encounter. If everything else fails, then the function defining an ellipsis as its parameter list is selected.

Usually that’s not a productive alternative, but in the current situation it is exactly what is needed. When confronted with two candidate functions, one of which defines an ellipsis parameter, the compiler selects the function defining the ellipsis parameter only if the alternative(s) can’t be used.

Following the above reasoning an alternative function `test(...)` is declared as well. This alternate function does not return a `Convertible` value but a `NotConvertible` value:

```cpp
NotConvertible test(...);
```
If `test`'s argument is of type `T` and if `T` can be converted to `U` then `test`'s return type is `Convertible`. Otherwise `NotConvertible` is returned.

This situation clearly shows similarities with the situation encountered in section 23.6.1 where the value `isClass` had to be determined compile time. Here two related problems must be solved:

- **how do we obtain a `T` argument?** This is more difficult than might be expected at first sight as it might not be possible to define a `T`. If type `T` does not define any constructor then no `T` object can be defined.

- **how can `Convertible` be distinguished from `NotConvertible`?**

The first problem is solved by realizing that no `T` needs to be defined. After all, the intent is to decide *compile-time* whether a type is convertible and not to define a `T` value or object. Defining objects is not a compile-time but a *run-time* matter.

By simply *declaring* a function returning a `T` we can tell the compiler where it should assume a `T`:

```cpp
T makeT();
```

This mysterious function has the magical power of enticing the compiler into thinking that a `T` object comes out of it. However, this function needs a small modification before it will actually suit our needs. If, for whatever reason, `T` happens to be an array then the compiler will choke on `T makeT()` as functions cannot return arrays. This, however, is easily solved, as functions *can* return *references* to arrays. So the above declaration is changed into:

```cpp
T const &makeT();
```

Next we pass a `T const &` to `test`: following code:

```cpp
test(makeT())
```

Now that the compiler sees `test` being called with a `T const &` argument it decides that its return value is `Convertible` if a conversion is in fact possible. Otherwise it decides that its return value is `NotConvertible` (as the compiler, in that case, selected `test(...)`).

The second problem, distinguishing `Convertible` from `NotConvertible` is solved exactly the way `isClass` could be determined in section 23.6.1, viz. by making their sizes different. Having done so the following expression determines whether `T` is convertible from `U` or not:

```cpp
isConvertible = sizeof(test(makeT())) == sizeof(Convertible);
```

By using `char` for `Convertible` and `Char2` (cf. section 23.6.1) for `NotConvertible` the distinction can be made.

The above can be summarized in a class template `LconvertibleToR`, having two template type parameters:

```cpp
template <typename T, typename U>
class LconvertibleToR
{
    struct Char2
```
As the class template deletes its copy constructor no object can be created. Only its enum values can be interrogated. The next example writes 1 0 1 0 when run from a main function:

cout <<
LconvertibleToR<ofstream, ostream>::yes << " " <<
LconvertibleToR<ostream, ofstream>::yes << " " <<
LconvertibleToR<int, double>::yes << " " <<
LconvertibleToR<int, string>::yes <<
"\n";

23.8.3.1 Determining inheritance

Now that it is possible to determine type convertibility, it’s easy to determine whether a type Base is a (public) base class of a type Derived.

Inheritance is determined by inspecting convertibility of (const) pointers. Derived const * can be converted to Base const * if

- both types are identical;
- Base is a public and unambiguous base class of Derived;
- and (usually not intended) if Base is void.

Assuming the last conversion isn’t used inheritance can be determined using the following trait class LBaseRDerived. LBaseRDerived provides an enum yes which is 1 if the left type is a base class of the right type and both types are different:

```cpp
template <typename Base, typename Derived>
struct LBaseRDerived
{
```
If code should not consider a class to be its own base class, then the trait class `LBaseRtrulyDerived` can be used to perform a strict test. This trait class adds a test for type-equality:

```cpp
template <typename Base, typename Derived>
struct LBaseRtrulyDerived
{
    LBaseRtrulyDerived(LBaseRtrulyDerived const &) = delete;
    enum {
        yes =
            LBaseRDerived<Base, Derived>::yes &&
            not LconvertibleToR<Base const *, Derived const *>::sameType
    };
};
```

Example: the next statement displays 1: 0, 2: 1, 3: 0, 4: 1, 5: 0 when executed from a `main` function:

```cpp
cout << "\n" <<
"1: " << LBaseRDerived<ofstream, ostream>::yes << ", " <<
"2: " << LBaseRDerived<ostream, ofstream>::yes << ", " <<
"3: " << LBaseRDerived<void, ofstream>::yes "), " " <<
"4: " << LBaseRDerived<ostream, ostream>::yes << ", " <<
"5: " << LBaseRtrulyDerived<ostream, ostream>::yes <<
"\n";
```

### 23.9 Template TypeList processing

This section serves two purposes. It illustrates capabilities of the various template metaprogramming techniques, which can be used as a source of inspiration when developing your own templates; and it offers a concrete example, illustrating some of the power offered by these techniques.

This section itself was inspired by Andrei Alexandrescu’s (2001) book *Modern C++ design*. It diverges from Alexandrescu’s book in its use of variadic templates which were not yet available when he wrote his book. Even so, the algorithms used by Alexandrescu are still useful when using variadic templates.

The C++11 standard offers the tuple to store and retrieve values of multiple types. Here the focus is merely on processing types. A simple struct `TypeList` is going to be used as our working horse for the upcoming subsections. Here is its definition:

```cpp
template <typename ...Types>
struct TypeList
{
    TypeList(TypeList const &) = delete;
    enum { size = sizeof ...(Types) };  
};

A typelist allows us to store any number of types. Here is an example storing the three types char, short, int in a TypeList:

TypeList<char, short, int>

23.9.1 The length of a TypeList

As the number of types in a parameter pack may be obtained using the sizeof operator (cf. section 22.5) it is easy to obtain the number of types that were specified with a certain TypeList. For example, the following statement displays the value 3:

std::cout << TypeList<int, char, bool>::size << '\n';

However, it’s illustrative to see how the number of types specified with a TypeList could be determined if sizeof hadn’t been available.

To obtain the number of types that were specified with a TypeList the following algorithm is used:

- If the TypeList contains no types, its size equals zero;
- If the TypeList contains types, its size equals 1 plus the number of types that follow its first type.

The algorithm uses recursion to define the length of a TypeList. In executable C++ recursion could also be used in comparable situations. For example recursion can be used to determine the length of an NTBS:

```cpp
size_t c_length(char const *cp)
{
    return *cp == 0 ? 0 : 1 + c_length(cp + 1);
}
```

While C++ functions usually use iteration rather than recursion, iteration is not available to template meta programming algorithms. In template meta programming repetition must be implemented using recursion. Furthermore, while C++ run-time code may use conditions to decide whether or not to start the next recursion template meta programming cannot do so. Template meta programming algorithms must resort to (partial) specializations. The specializations are used to select alternatives.

The number of types that are specified in a TypeList can be computed using the following alternate implementation of TypeList, using a generic struct declaration and two specialization for the empty and non-empty TypeList (cf. the above description of the algorithm):

```cpp
template <typename ...Types>
struct TypeList;
```
23.9. TEMPLATE TYPEDLIST PROCESSING

template <typename Head, typename ...Tail>
struct TypeList
{
    enum { size = 1 + TypeList<Tail...>::size };
};
template <>
struct TypeList<>
{
    enum { size = 0 };
};

23.9.2 Searching a TypeList

To determine whether a particular type (called SearchType below) is present in a given TypeList, an algorithm is used that either defines ‘index’ as -1 (if SearchType is not an element of the TypeList) or it defines ‘index’ as the index of the first occurrence of SearchType in the TypeList. The following algorithm is used:

- If the TypeList is empty, ‘index’ is -1;
- If the TypeList’s first element equals SearchType, ‘index’ is 0;
- Otherwise ‘index’ is:
  - -1 if searching for SearchType in TypeList’s tail results in ‘index’ == -1;
  - Otherwise (SearchType was found in TypeList’s tail) index is set to 1 + the index obtained when searching for SearchType in the TypeList’s tail.

The algorithm is implemented using a variadic template struct ListSearch expecting a parameter pack:

template <typename ...Types>
struct ListSearch
{
    ListSearch(ListSearch const &p) = delete;
};

Specializations handle the alternatives mentioned with the algorithm:

- If TypeList is empty, ‘index’ is -1:
  
  template <typename SearchType>
  struct ListSearch<SearchType, TypeList<>>
  {
      ListSearch(ListSearch const &p) = delete;
      enum { index = -1 };
  };

- If TypeList’s head equals SearchType, ‘index’ is 0. Note that SearchType is explicitly mentioned as the TypeList’s first element:

  template <typename SearchType, typename ...Tail>
  struct ListSearch<SearchType, TypeList<Tail...>>
  {
      ListSearch(ListSearch const &p) = delete;
      enum { index = 0 };
  };

  template <typename SearchType, typename ...Tail>
CHAPTER 23. ADVANCED TEMPLATE USE

struct ListSearch<SearchTree, TypeList<SearchTree, Tail...>>
{
    ListSearch(ListSearch const &) = delete;
    enum { index = 0 };
};

- Otherwise a search is performed on the TypeList’s tail. The index value returned by this
  search is stored in a tmp enum value, which is then used to determine index’s value.

template <typename SearchType, typename Head, typename ...Tail>
struct ListSearch<SearchTree, TypeList<Head, Tail...>>
{
    ListSearch(ListSearch const &) = delete;
    enum { tmp = ListSearch<SearchTree, TypeList<Tail...>>::index };  
    enum { index = tmp == -1 ? -1 : 1 + tmp }; 
};

Here is an example showing how ListSearch can be used:

std::cout <<
    ListSearch<char, TypeList<int, char, bool>>::index << "\n"
    ListSearch<float, TypeList<int, char, bool>>::index << "\n";

23.9.3 Selecting from a TypeList

The inverse operation of determining the index of a certain type in a TypeList is retrieving the
type given its index. This inverse operation is the topic of this section.

The algorithm is implemented using a struct TypeAt. TypeAt uses a typedef to define the type
matching a given index. But the index might be out of bounds. In that case we have several options:

- Use a static_assert to stop the compilation. This is an appropriate action if the index should
  simply not be out of bounds;

- Define a local type (e.g., Null) that should not be used as a type in the TypeList. This type
  is going to be returned when the index is out of bounds. Using this local type as one of the types
  in a TypeList is considered an error as its would conflict with the special meaning of Null as
  the type returned at an invalid index.
  To prevent Null from being returned by TypeAt a static_assert is used to catch the Null
type when it is encountered while evaluating TypeAt;

- The struct TypeAt may define an enum value validIndex set to true if the index was valid
  and set to false if not.

The first alternative is implemented below. The other alternatives are not difficult to implement and
are left as exercises for the reader. Here’s how TypeAt works:

- The foundation consists of a variadic template struct TypeAt, expecting an index and a
  TypeList:

    template <size_t index, typename Typelist>
    struct TypeAt;
• If the typelist is empty a static_assert ends the compilation

```cpp
template <size_t index>
struct TypeAt<index, TypeList<>>
{
    static_assert(index < 0, "TypeAt index out of bounds");
    typedef TypeAt Type;
};
```

• If the search index equals 0, define Type as the first type in the TypeList:

```cpp
template <typename Head, typename ...Tail>
struct TypeAt<0, TypeList<Head, Tail...>>
{
    typedef Head Type;
};
```

• Otherwise, Type is defined as Type defined by TypeAt<index - 1> operating on the TypeList’s tail:

```cpp
template <size_t index, typename Head, typename ...Tail>
struct TypeAt<index, TypeList<Head, Tail...>>
{
    typedef typename TypeAt<index - 1, TypeList<Tail...>>::Type Type;
};
```

Here is how typeAt can be used. Uncommenting the first variable definition causes a TypeAt index out of bounds compilation error:

```cpp
typedef TypeList<int, char, bool> list3;
//  TypeAt<3, list3>::Type invalid;
TypeAt<0, list3>::Type intVariable = 13;
TypeAt<2, list3>::Type boolVariable = true;

cout << "The size of the first type is " <<
    sizeof(TypeAt<0, list3>::Type) << ", " <<
    "the size of the third type is " <<
    sizeof(TypeAt<2, list3>::Type) << ";n";

if (typeid(TypeAt<1, list3>::Type) == typeid(char))
cout << "The typelist’s 2nd type is char\n";

if (typeid(TypeAt<2, list3>::Type) != typeid(char))
cout << "The typelist’s 3nd type is not char\n";
```

### 23.9.4 Prefixing/Appending to a TypeList

Prepending or appending a type to a TypeList is easy and doesn’t require recursive template meta programs. Two variadic template structs Append and Prefix and two specializations are all it takes.
Here are the declarations of the two variadic template structs:

```cpp
template <typename ...Types>
struct Append;

template <typename ...Types>
struct Prefix;
```

To append or prefix a new type to a typelist, specializations expect a typelist and a type to add. Then, they simply define a new TypeList also including the new type. The Append specialization shows that a template pack does not have to be used as the first argument when defining another variadic template type:

```cpp
template <typename NewType, typename ...Types>
struct Append<TypeList<Types...>, NewType>
{
    typedef TypeList<Types..., NewType> List;
};

template <typename NewType, typename ...Types>
struct Prefix<NewType, TypeList<Types...>>
{
    typedef TypeList<NewType, Types...> List;
};
```

### 23.9.5 Erasing from a TypeList

It is also possible to erase types from a TypeList. Again, there are several possibilities, each resulting in a different algorithm.

- The type to erase is specified, and the first occurrence of that type is removed from the TypeList;
- The index of the type to erase is specified, and the type at that index position is erased from the TypeList;
- The type to erase is specified, and all occurrences of that type are removed from the TypeList.
- As a variant of erasure: we may want to erase all multiply occurring types in a TypeList, keeping each type only once.

Doubtlessly there are other ways of erasing types from a TypeList. Which ones are eventually implemented depends of course on the circumstances. As template meta programming is very powerful most if not all algorithms can probably be implemented. As an illustration of how to erase types from a TypeList the above-mentioned algorithms are now developed in the upcoming subsections.

#### 23.9.5.1 Erasing the first occurrence

To erase the first occurrence of a specified EraseType from a TypeList a recursive algorithm is used once again. The template meta program uses a generic Erase struct and several specializations. The specializations define a type List containing the resulting TypeList after the erasure. Here is the algorithm:
23.9. TEMPLATE TYPELIST PROCESSING

- The foundation of the algorithm consists of a struct template `Erase` expecting the type to erase and a `TypeList`:

  ```cpp
  template <typename EraseType, typename TypeList>
  struct Erase;
  ```

- If the typelist is empty, there's nothing to erase, and an empty `TypeList` results:

  ```cpp
  template <typename EraseType>
  struct Erase<EraseType, TypeList<>>
  {
    typedef TypeList<> List;
  };
  ```

- If the `TypeList`'s head matches the type to erase, then `List` becomes a `TypeList` containing the original `TypeList`'s tail types:

  ```cpp
  template <typename EraseType, typename ...Tail>
  struct Erase<EraseType, TypeList<EraseType, Tail...>>
  {
    typedef TypeList<Tail...> List;
  };
  ```

- In all other cases the erase operation is applied to the `TypeList`'s tail. This results in a `TypeList` to which the original `TypeList`'s head must be prefixed. The `TypeList` returned by the prefix operation is then returned as `Erase::List`:

  ```cpp
  template <typename EraseType, typename Head, typename ...Tail>
  struct Erase<EraseType, TypeList<Head, Tail...>>
  {
    typedef typename
    Prefix<Head,
      typename Erase<EraseType, TypeList<Tail...>>::List
    >::List List;
  };
  ```

Here is a statement showing how `Erase` can be used:

```
cout <<
    Erase<int, TypeList<char, double, int>>::List::size << '\n' <<
    Erase<char, TypeList<int>>::List::size << '\n' <<
    Erase<int, TypeList<int>>::List::size << '\n' <<
    Erase<int, TypeList>>::List::size << "\n";
```

23.9.5.2 Erasing a type by its index

To erase a type from a `TypeList` by its index we again use a recursive template meta program. `EraseIdx` expects a `size_t` index value and a `TypeList` from which its `idx`\textsuperscript{th} (0-based) type must be erased. `EraseIdx` defines the `TypeList` containing the resulting `TypeList`. Here is the algorithm:

- The foundation of the algorithm consists of a struct template `EraseIdx` expecting the index of the type to erase and a `TypeList`:

  ```cpp
  template <size_t idx, typename TypeList>
  struct EraseIdx;
  ```
• If the typelist is empty, there's nothing to erase, and an empty TypeList results:

```cpp
template <size_t idx>
struct EraseIdx<idx, TypeList<>>
{
    typedef TypeList<> List;
};
```

• The recursion otherwise ends once `idx` becomes 0. At that point the TypeList's first type is ignored and `List` is initialized to a TypeList containing the types in the original TypeList's tail:

```cpp
template <typename EraseType, typename ...Tail>
struct EraseIdx<0, TypeList<EraseType, Tail...>>
{
    typedef TypeList<Tail...> List;
};
```

• In all other cases `EraseIdx` is applied to the TypeList's tail, providing it with a decremented value of `idx`. To the resulting TypeList the original TypeList's head is prefixed. The TypeList returned by the prefix operation is then returned as `EraseIdx::List`:

```cpp
template <size_t idx, typename Head, typename ...Tail>
struct EraseIdx<idx, TypeList<Head, Tail...>>
{
    typedef typename Prefix<
        Head,
        typename EraseIdx<idx - 1, TypeList<Tail...>>::List
    >::List List;
};
```

Here is a statement showing how `EraseIdx` can be used:

```cpp
if
(
    typeid(TypeAt<2, 
        EraseIdx<1, 
            TypeList<int, char, size_t, double, int>>::List
    >::Type
) == typeid(double)
)
    cout << "the third type is now a double\n";
```

### 23.9.5.3 Erasing all occurrences of a type

Erasing all types `EraseType` from a TypeList can easily be accomplished by applying the erasure procedure not only to the head of the TypeList but also to the TypeList's tail.

Here is the algorithm, described in a slightly different order than `Erase`'s algorithm:

• If the TypeList is empty, there's nothing to erase, and an empty TypeList results. This is exactly what we do with `Erase`, so we can use inheritance to prevent us from having to duplicate elements of a template meta program:
23.9. TEMPLATE TYPELIST PROCESSING

```cpp
template <size_t idx>
struct EraseIdx<idx, TypeList<>>
{
    typedef TypeList<> List;
};
```

- The foundation of the algorithm is therefore a struct template `EraseAll` expecting the type to erase and a `TypeList` that is derived from `Erase`, thus already offering the empty `TypeList` handling specialization:

```cpp
template <typename EraseType, typename TypeList>
struct EraseAll: public Erase<EraseType, TypeList>
{
};
```

- If `TypeList`'s head matches `EraseType` `EraseAll` is also applied to the `TypeList`'s tail, thus removing all occurrences of `EraseType` from `TypeList`:

```cpp
template <typename EraseType, typename ...Tail>
struct EraseAll<EraseType, TypeList<EraseType, Tail...>>
{
    typedef typename EraseAll<EraseType, TypeList<Tail...>>::List List;
};
```

- In all other cases (i.e., `TypeList`'s head does not match `EraseType`) `EraseAll` is applied to the `TypeList`'s tail. The returned `TypeList` consists of the original `TypeList`'s initial type and the types of the `TypeList` returned by the recursive `EraseAll` call:

```cpp
template <typename EraseType, typename Head, typename ...Tail>
struct EraseAll<EraseType, TypeList<Head, Tail...>>
{
    typedef typename Prefix<
        Head,
        typename EraseAll<EraseType, TypeList<Tail...>>::List
    >::List List;
};
```

Here is a statement showing how `EraseAll` can be used:

```cpp
cout <<
    "After erasing size_t from "
    "TypeList<char, int, size_t, double, size_t>\n"
    "it contains " <<
    EraseAll<size_t,
        TypeList<char, int, size_t, double, size_t>
    >::List::size << " types\n";
```

23.9.5.4 Erasing duplicates

To remove all duplicates from a `TypeList` all the `TypeList`'s first elements must be erased from the `TypeList`'s tail, applying the procedure recursively to the `TypeList`'s tail. The algorithm, outlined below, merely expects a `TypeList`:
• First, the generic `EraseDup` struct template is declared. `EraseDup` structures define a type `List` representing the `TypeList` that they generate. `EraseDup` calls expect a `TypeList` as their template type parameters:

```cpp
template <typename TypeList>
struct EraseDup;
```

• If the `TypeList` is empty it can be returned empty and we’re done:

```cpp
template <>
struct EraseDup<TypeList<>>
{
    typedef TypeList<> List;
};
```

• In all other cases

  - `EraseDup` is first applied to the original `TypeList`’s tail. By definition this results in a `TypeList` from which all duplicates have been removed;
  - The `TypeList` returned by the previous step might contain the original `TypeList`’s initial type. If so, it is removed by applying `Erase` on the returned `TypeList`, specifying the original `TypeList`’s initial type as the type to remove;
  - The returned `TypeList` consists of the original `TypeList`’s initial type to which the types of the `TypeList` produced by the previous step are appended.

This specialization is implemented like this:

```cpp
template <typename Head, typename ...Tail>
struct EraseDup<TypeList<Head, Tail...>>
{
    typedef typename EraseDup<TypeList<Tail...>>::List UniqueTail;
    typedef typename Erase<Head, UniqueTail>::List NewTail;

    typedef typename Prefix<Head, NewTail>::List List;
};
```

Here is an example showing how `EraseDup` can be used:

```cpp
cout <<
"After erasing duplicates from 
" "TypeList<double, char, int, size_t, int, double, size_t>\n"
"it contains " <<
EraseDup<
    TypeList<double, char, int, size_t, int, double, size_t>
>:\:\List::size << " types\n";
```

### 23.10 Using a TypeList

In the previous sections the definition and some of the features of typelists were discussed. Most C++ programmers consider typelists both exciting and an intellectual challenge, honing their skills in the area of recursive programming.
But there’s more to typelist than a mere intellectual challenge. In the final sections of this chapter the following topics are covered:

- Creating classes from a typelist.
  Here the aim is to construct a new class consisting of instantiations of an existing basic template for each of the types mentioned in a provided typelist;

- Accessing data members from the thus constructed conglomerate class by index, rather than name.

Again, much of the material covered by these sections was inspired by Alexandrescu’s (2001) book, this time combined with the features offered by the C++11 standard.

### 23.10.1 The Wrap and Multi class templates

To illustrate template meta programming concepts the template class `Multi` is now developed. The class template `Multi` creates a new class from a template template parameter `Policy` defining the data storage policy and a series of types from which `Multi` is eventually derived. It does so by passing its template parameters to its base class `MultiBase` that in turn creates a final class inheritance tree. Since we don’t know how many types are going to be used `Multi` is defined as a variadic class template using a template pack `...Types`.

In fact, the types that are specified with `Multi` aren’t that interesting. They primarily serve to ‘seed’ the class `Policy`. Therefore, rather than forwarding `Multi`’s types to `MultiBase` they are passed to `Policy` and the sequence of `Policy<Type>` types is then forwarded to `MultiBase`. `Multi`’s constructor expects initialization values for its various `Policy<Type>`s which are perfectly forwarded to `MultiBase`.

The class `Multi` (implementing its constructor in-class to save some space) shows how a template pack can be wrapped into a policy. Here is `Multi`’s definition:

```cpp
template <template <typename> class Policy, typename ...Types>
struct Multi: public MultiBase<0, Policy<Types>...>
{
    typedef TypeList<Types...> PlainTypes;
    typedef MultiBase<0, Policy<Types>...> Base;

    enum { size = PlainTypes::size };

    Multi(Policy<Types> &&...types)
    :
        MultiBase<0, Policy<Types>...>{
            std::forward<Policy<Types>>(types)...
        }
    {};
};
```

Unfortunately, the design as described contains some flaws.

- As the `Policy` template template parameter is defined as `template <typename> class Policy` it can only accept policies expecting one type argument. Contrary to this, `std::vector` is a template expecting two template arguments, the second one defining the allocation scheme used by `std::vector`. This allocation scheme is hardly ever changed, and
most applications merely define objects of types like vector<int>, vector<string> etc.
Template template parameters must, however, be specified with the correct number and types of required template parameters so vector can't be specified as a policy for Multi. This can be solved by wrapping a more complex template in a simpler wrapper template, like so:

```cpp
template <class Type>
struct Vector: public std::vector<Type>
{
    Vector(std::initializer_list<Type> iniValues)
    :
        std::vector<Type>(iniValues)
    {}
};
```

Now Vector provides std::vector's second parameter using its default template argument. Alternatively, a template using declaration could be used.

- If the TypeList contains two types like int and double and the policy class is Vector, then the MultiBase class eventually inherits from vector<int> and vector<double>. But if the TypeList contains identical types, like two int type specifications MultiBase would inherit from two vector<int> classes. Classes cannot be derived from identical base classes as that would make it impossible to distinguish among their members. Regarding this, Alexandrescu (2001) writes (p.67):

  There is one major source of annoyance...: you cannot use it when you have duplicate types in your TypeList.
  .... There is no easy way to solve the ambiguity, [as the eventually derived class /FBB] ends up inheriting [the same base class /FBB] twice.

There is a way around the problem of duplicate base class types. If instead of inheriting directly from base classes these base classes are first wrapped in unique type defining classes, then these unique classes can be used to access the base classes using principles of inheritance. As these unique type-defining wrapper classes are merely classes that are derived from the ‘real’ base classes they inherit (and thus: offer) the functionality of their base classes. A unique type defining wrapper class can be designed after the class IntType, defined earlier. The wrapper class we're looking combines class derivation with the uniqueness offered by IntType. The class template UWrap has two template parameters: one non-type parameter idx and one type parameter. By ensuring that each UWrap definition uses a unique idx value unique class types are created. These unique class types are then used as base classes of the derived class MultiBase:

```cpp
template <size_t nr, typename Type>
struct UWrap: public Type
{
    UWrap(Type const &type)
    :
        Type(type)
    {}
};
```

Using UWrap it's easy to distinguish, e.g., two vector<int> classes: UWrap<0, vector<int>> could refer to the first vector<int>, UWrap<1, vector<int>> to the second vector.

Uniqueness of the various UWrap types is assured by the class template MultiBase as discussed in the next section.
23.10. USING A TYPELIST

It must also be possible to initialize a `Multi` class object. Its constructor therefore expects the initialization values for all its `Policy` values. So if a `Multi` is defined for `Vector`, `int`, `string` then its constructor can receive the matching initialization values. E.g.,

```cpp
Multi<Vector, int, string> mvis({1, 2, 3}, {"one", "two", "three"});
```

23.10.2 The MultiBase class template

The class template `MultiBase` is `Multi`’s base class. It defines a class that, eventually, is derived from the list of `Policy` types that, in turn, were created by `Multi` using any additional types that were passed to it.

`MultiBase` itself has no concept of a `Policy`. To `MultiBase` the world appears to consist of a simple template pack whose types are used to define a class from. In addition to the `PolicyTypes` template pack, `MultiBase` also defines a `size_t nr` non-type parameter that is used to create unique `UWrap` types. Here is `MultiBase`’s generic class declaration:

```cpp
template <size_t nr, typename ...PolicyTypes>
struct MultiBase;
```

Two specializations handle all possible `MultiBase` invocations. One specialization is a recursive template. This template handles the first type of `MultiBase`’s template parameter pack and recursively uses itself to handle the remaining types. The second specialization is invoked once the template parameter pack is exhausted and does nothing. Here is the definition of the latter specialization:

```cpp
template <size_t nr>
struct MultiBase<nr> {
};
```

The recursively defined specialization is the interesting one. It performs the following tasks:

- It is derived from a unique `UWrap` type. The uniqueness is guaranteed by using `MultiBase`’s `nr` parameter when defining `UWrap`. In addition to `nr` the `UWrap` class receives the first type of the template parameter pack made available to `MultiBase`;
- It is also recursively derived from itself. The recursive `MultiBase` type is defined using as its first template argument an incremented `nr` value (thus ensuring the uniqueness of the `UWrap` types defined by recursive `MultiBase` types). Its second template argument is the tail of the template parameter pack made available to `MultiBase`.

An illustration showing the layout of the `MultiBase` class hierarchy is provided in figure 23.1.

`MultiBase`’s constructor simply receives the initialization values that were (originally) passed to the `Multi` object. Perfect forwarding is used to accomplish this. `MultiBase`’s constructor passes its first parameter value to its `UWrap` base class, also using perfect forwarding. `MultiBase`’s recursive definition is:

```cpp
template <size_t nr, typename PolicyT1, typename ...PolicyTypes>
struct MultiBase<nr, PolicyT1, PolicyTypes...> :
    public UWrap<nr, PolicyT1>,
```
public MultiBase<nr + 1, PolicyTypes...> 
{
    typedef PolicyT1 Type;
    typedef MultiBase<nr + 1, typename PolicyTypes...> Base;

    MultiBase(PolicyT1 && policyt1, PolicyTypes &&...policytypes)
    :
        UWrap<nr, PolicyT1>{std::forward<PolicyT1>(policyt1)},
        MultiBase<nr + 1, PolicyTypes...>{
            std::forward<PolicyTypes>(policytypes)...}
    
};

\section*{23.10.3 Support templates}

The Multi class template defines PlainTypes as the TypeList holding all the types of its parameter pack. Each MultiBase derived from a UWrap type also defines a type Type representing the policy type that was used to define the UWrap type and a type Base representing the type of its nested MultiBase class.

These three type definitions allow us to access the types from which the Multi object was created as well as the values of those types.

The class template typeAt, is a pure template meta program class template (it has no run-time executable code). It expects a size_t idx template argument specifying the index of the policy type in a Multi type object as well as a Multi class type. It defines the type Type as the Type defined by Multi's MultiBase<idx, ...> base class. Example:

    typeAt<0, Multi<Vector, int, double>::Type // Type is vector<int>

The class template typeAt defines (and uses) a nested class template PolType doing all the work. PolType's generic definition specifies two template parameters: an index used to specify the index of the requested type and a typename initialized by a MultiBase type argument. PolType's re-
cursive definition recursively reduces its index non-type parameter, passing the next base class in
MultiBase's inheritance tree to the recursive call. As PolType eventually defines the type Type
to be the requested policy type the recursive definition defines its Type as the type defined by the
recursive call. The final (non-recursive) specialization defines the initial policy type of the MultiBase
type as Type. Here is typeAt's definition:

```cpp
template <size_t index, typename Multi>
class typeAt
{
    template <size_t idx, typename MultiBase>
    struct PolType;

    template <size_t idx,
               size_t nr, typename PolicyT1, typename ...PolicyTypes>
    struct PolType<idx, MultiBase<nr, PolicyT1, PolicyTypes...>>
    {
        typedef typename PolType<
            idx - 1, MultiBase<nr + 1, PolicyTypes...>>::Type Type;
    }

    template <size_t nr, typename PolicyT1, typename ...PolicyTypes>
    struct PolType<0, MultiBase<nr, PolicyT1, PolicyTypes...>>
    {
        typedef PolicyT1 Type;
    }

    public:
    typeAt(typeAt const &) = delete;
    typedef typename PolType<index, typename Multi::Base>::Type Type;
};
```

The types specified by Multi's parameter pack can also be retrieved using a second helper class
template: plainTypeAt. Example:

```cpp
plainTypeAt<0, Multi<Vector, int, double>>::Type // Type is int
```

The class template plainTypeAt uses a comparable (but simpler) implementation than typeAt.
It is also a pure template meta program class template defining a nested class template At. At is
implemented like typeAt but it visits the types of the original template pack that was passed to
Multi, and made available by Multi as its PlainTypes type. Here is plainTypeAt's definition:

```cpp
template <size_t index, typename Multi>
class plainTypeAt
{
    template <size_t idx, typename List>
    struct At;

    template <size_t idx, typename Head, typename ...Tail>
    struct At<idx, TypeList<Head, Tail...>>
    {
        typedef typename At<idx - 1, TypeList<Tail...>>::Type Type;
    }

    template <typename Head, typename ...Tail>
```
Arguably the neatest support template is `get`. This is a function template defining `size_t idx` as its first template parameter and `typename Multi` as its second template parameter. The function template `get` defines one function parameter: a reference to a `Multi`, so it can deduce `Multi`'s type by itself. Knowing that it's a `Multi`, we reason that it is also a `UWrap<nr, PolicyType>` and therefore also a `PolicyType`, as the latter class is defined as a base class of `UWrap`.

Since class type objects can initialize references to their base classes the `PolicyType &` can be initialized by an appropriate `UWrap` reference, which in turn can be initialized by a `Multi` object. Since we can determine `PolicyType` using `TypeAt` (note that evaluating `typename typeAt<idx, Multi>::Type` is a purely compile-time matter), the `get` function can very well be implemented `inline` by a single `return` statement:

```cpp
template <size_t idx, typename Multi>
inline typename typeAt<idx, Multi>::Type &get(Multi &multi)
{
    return static_cast<
        UWrap<idx, typename typeAt<idx, Multi>::Type> &>(multi);
}
```

The intermediate `UWrap` cast is required to disambiguate between identical policy types (like two `vector<int>` types). As `UWrap` is uniquely determined by its `nr` template argument and this is the number argument that is passed to `get` ambiguities can easily be prevented.

### 23.10.4 Using Multi

Now that `Multi` and its support templates have been developed, how can a `Multi` be used?

A word of warning is in place. To reduce the size of the developed classes they were designed in a minimalist way. For example, the `get` function template cannot be used with `Multi const` objects and there is no default, or move constructor available for `Multi` types. `Multi` was designed to illustrate some of the possibilities of template meta programming and hopefully `Multi`'s implementation served that purpose well. But can it be used? If so, how?

This section provides some annotated examples. They may be concatenated to define a series of statements that could be placed in a `main` function's body, which would result in a working program.

- A simple `Policy` could be defined:

```cpp
template <typename Type>
struct Policy
{
    Type d_type;
    Policy(Type &&type) = default;
};
```
: d_type(std::forward<Type>(type))
{};
}
)

Policy defines a data member and it can be used to define Multi objects:

Multi<Policy, string> ms(Policy<string>("hello");
Multi<Policy, string, string> ms2s(Policy<string>("hello"),
Policy<string>("world");

typedef Multi<Policy, string, int> MPSI;
MPSI mpsi(string("hello"), 4);

• To obtain the number of types defined by a Multi class or object either use the ::size enum value (using the Multi class) or the .size member (using the Multi object):

    cout << "There are " << MPSI::size << " types in MPSI\n"
    "There are " << mpsi.size << " types in mpsi\n";

• Variables of constituting types can be defined using plainTypeAt:

    plainTypeAt<0, MPSI>::Type sx = "String type";
    plainTypeAt<1, MPSI>::Type ix = 12;

• Raw static casts can be used to obtain the constituent type:

    cout << static_cast<Policy<string> &>(mpsi).d_type << 'n' <<
    static_cast<Policy<int> &>(mpsi).d_type << 'n';

• However, this won't work when the template parameter pack contains identical types, as a cast can't distinguish between identical Policy<Type> types. In that case get still works fine:

    typedef Multi<Policy, int, int> MPII;
    MPII mpii(4, 18);
    cout << get<0>(mpii).d_type << ' ' << get<1>(mpii).d_type << 'n';

• Here is an example wrapping a std::vector in a Vector:

    typedef Multi<Vector, int, double> MVID;
    MVID mi({1, 2, 3}, {1.2, 3.4, 5.6, 7.8});

• Such a vector can be defined by its Multi type:

    typeAt<0, Multi<Vector, int>>::Type vi = {1, 2, 3};

• Knowing that a Vector is a std::vector, the reference returned by get support index opera-
    tors that can be used as left hand side or right hand side operands:

    cout << get<0>(mi)[2] << 'n';
    get<1>(mi)[3] = get<0>(mi)[0];
    cout << get<1>(mi)[3] << 'n';
Chapter 24

Concrete Examples

In this chapter concrete examples of C++ programs, classes and templates are presented. Topics covered by the C++ Annotations such as virtual functions, static members, etc. are illustrated in this chapter. The examples roughly follow the organization of earlier chapters.

As an additional topic, not just providing examples of C++ the subjects of scanner and parser generators are covered. We show how these tools may be used in C++ programs. These additional examples assume a certain familiarity with the concepts underlying these tools, like grammars, parse-trees and parse-tree decoration. Once the input for a program exceeds a certain level of complexity, it’s attractive to use scanner- and parser-generators to create the code doing the actual input processing. One of the examples in this chapter describes the usage of these tools in a C++ environment.

24.1 Using file descriptors with ‘streambuf’ classes

24.1.1 Classes for output operations

Reading and writing from and to file descriptors are not part of the C++ standard. But on most operating systems file descriptors are available and can be considered a device. It seems natural to use the class std::streambuf as the starting point for constructing classes interfacing such file descriptor devices.

Below we’ll construct classes that can be used to write to a device given its file descriptor. The devices may be files, but they could also be pipes or sockets. Section 24.1.2 covers reading from such devices; section 24.2.3 reconsidered redirection, discussed earlier in section 6.6.1.

Using the streambuf class as a base class it is relatively easy to design classes for output operations. The only member function that must be overridden is the (virtual) member int streambuf::overflow(int c). This member’s responsibility is to write characters to the device. If fd is an output file descriptor and if output should not be buffered then the member overflow() can simply be implemented as:

```cpp
class UnbufferedFD: public std::streambuf
{
    public:
        virtual int overflow(int c);
    ...
```

877
int UnbufferedFD::overflow(int c)
{
    if (c != EOF)
    {
        if (write(d_fd, &c, 1) != 1)
            return EOF;
    }
    return c;
}

The argument received by overflow is either written to the file descriptor (and returned from overflow), or EOF is returned.

This simple function does not use output buffering. For various reasons, using a buffer is usually a good idea (see also the next section).

When output buffering is used, the overflow member is a bit more complex as it is only called when the buffer is full. Once the buffer is full, we first have to flush the buffer. Flushing the buffer is the responsibility of the (virtual) function streambuf::sync. Since sync is a virtual function, classes derived from streambuf may redefine sync to flush a buffer streambuf itself doesn’t know about.

Overriding sync and using it in overflow is not all that has to be done. When the object of the class defining the buffer reaches the end of its lifetime the buffer may be only partially full. In that situation the buffer must also be flushed. This is easily done by simply calling sync from the class’s destructor.

Now that we’ve considered the consequences of using an output buffer, we’re almost ready to design our derived class. Several more features are added as well, though:

- First, we should allow the user of the class to specify the size of the output buffer.
- Second, it should be possible to construct an object of our class before the file descriptor is actually known. Later, in section 24.2 we’ll encounter a situation where this feature is actually used.

To save some space in the C++ Annotations, the successful completion of the functions designed here is not checked in the example code. In ‘real life’ implementations these checks should of course not be omitted. Our class OFdnStreambuf has the following characteristics:

- Its member functions use low-level functions operating on file descriptors. So apart from streambuf the <uninstd.h> header file must have been read by the compiler before its member functions can be compiled.
- The class is derived from std::streambuf.
- It defines three data members. These data members keep track of, respectively, the size of the buffer, the file descriptor, and the buffer itself. Here is the full class interface

```cpp
class OFdnStreambuf: public std::streambuf
{
    size_t d_bufsize;
    int    d_fd;
    char   *d_buffer;
```
24.1. USING FILE DESCRIPTORS WITH ‘STREAMBUF’ CLASSES

public:
  OFdnStreambuf();
  OFdnStreambuf(int fd, size_t bufsize = 1);
  virtual ~OFdnStreambuf();
  void open(int fd, size_t bufsize = 1);
private:
  virtual int sync();
  virtual int overflow(int c);
};

- Its default constructor merely initializes the buffer to 0. Slightly more interesting is its constructor expecting a file descriptor and a buffer size. This constructor passes its arguments on to the class’s open member (see below). Here are the constructors:

  inline OFdnStreambuf::OFdnStreambuf()
  :
    d_bufsize(0),
    d_buffer(0)
  {}
  inline OFdnStreambuf::OFdnStreambuf(int fd, size_t bufsize)
  {
    open(fd, bufsize);
  }

- The destructor calls sync, flushing any characters stored in the output buffer to the device. In implementations not using a buffer the destructor can be given a default implementation:

  inline OFdnStreambuf::~OFdnStreambuf()
  {
    if (d_buffer)
    {
      sync();
      delete[] d_buffer;
    }
  }

This implementation does not close the device. It is left as an exercise to the reader to change this class in such a way that the device is optionally closed (or optionally remains open). This approach was adopted by, e.g., the Bobcat library\(^1\). See also section 24.1.2.2.

- The open member initializes the buffer. Using streambuf::setp, the begin and end points of the buffer are defined. This is used by the streambuf base class to initialize streambuf::pbase, streambuf::pptr, and streambuf::epptr:

  inline void OFdnStreambuf::open(int fd, size_t bufsize)
  {
    d_fd = fd;
    d_bufsize = bufsize == 0 ? 1 : bufsize;

    d_buffer = new char[d_bufsize];
    setp(d_buffer, d_buffer + d_bufsize);
  }

\(^1\)http://bobcat.sourceforge.net
CHAPTER 24. CONCRETE EXAMPLES

- The member `sync` flushes the as yet unflushed contents of the buffer to the device. After the flush the buffer is reinitialized using `setp`. After successfully flushing the buffer `sync` returns 0:

  ```cpp
  inline int OFdnStreambuf::sync()
  {
    if (pptr() > pbase())
    {
      write(d_fd, d_buffer, pptr() - pbase());
      setp(d_buffer, d_buffer + d_bufsize);
    }
    return 0;
  }
  ```

- The member `streambuf::overflow` is also overridden. Since this member is called from the `streambuf` base class when the buffer is full it should first call `sync` to flush the buffer to the device. Next it should write the character `c` to the (now empty) buffer. The character `c` should be written using `pptr` and `streambuf::pbump`. Entering a character into the buffer should be implemented using available `streambuf` member functions, rather than 'by hand' as doing so might invalidate `streambuf`'s internal bookkeeping. Here is `overflow`'s implementation:

  ```cpp
  inline int OFdnStreambuf::overflow(int c)
  {
    sync();
    if (c != EOF)
    {
      *pptr() = c;
      pbump(1);
    }
    return c;
  }
  ```

The next program uses the `OFfdStreambuf` class to copy its standard input to file descriptor `STDOUT_FILENO`, which is the symbolic name of the file descriptor used for the standard output:

```cpp
#include <string>
#include <iostream>
#include <istream>
#include "fdout.h"
using namespace std;

int main(int argc, char **argv)
{
  OFdnStreambuf fds(STDOUT_FILENO, 500);
  ostream os(&fds);

  switch (argc)
  {
    case 1:
      for (string s; getline(cin, s); )
        os << s << '\n';
      os << "COPIED cin LINE BY LINE\n";
      break;
```
24.1. USING FILE DESCRIPTORS WITH ‘STREAMBUF’ CLASSES

```cpp
case 2:
    cin >> os.rdbuf(); // Alternatively, use: cin >> &fds;
    os << "COPIED cin BY EXTRACTING TO os.rdbuf()\n";
    break;

case 3:
    os << cin.rdbuf();
    os << "COPIED cin BY INSERTING cin.rdbuf() into os\n";
    break;
```

24.1.2 Classes for input operations

When classes for input operation are derived from `std::streambuf`, they should be provided with an input buffer of at least one character. The one-character input buffer allows for the use of the member functions `istream::putback` or `istream::ungetc`. Strictly speaking it is not necessary to implement a buffer in classes derived from `streambuf`. But using buffers in these classes is strongly advised. Their implementation is very simple and straightforward and the applicability of such classes is greatly improved. Therefore, all our classes derived from the class `streambuf` define a buffer of at least one character.

24.1.2.1 Using a one-character buffer

When deriving a class (e.g., `IFdStreambuf`) from `streambuf` using a buffer of one character, at least its member `streambuf::underflow` should be overridden, as this member eventually receives all requests for input. The member `streambuf::setg` is used to inform the `streambuf` base class of the size and location of the input buffer, so that it is able to set up its input buffer pointers accordingly. This ensures that `streambuf::eback`, `streambuf::gptr`, and `streambuf::egptr` return correct values.

The class `IFdStreambuf` is designed like this:

- Its member functions use low-level functions operating on file descriptors. Therefore, in addition to `streambuf`, the `<unistd.h>` header file must have been read by the compiler before its member functions can be compiled.

- Like most classes designed for input operations, this class is derived from `std::streambuf` as well.

- The class defines two data members, one of them a fixed-sized one character buffer. The data members are defined as `protected` data members so that derived classes (e.g., see section 24.1.2.3) can access them. Here is the full class interface:

```cpp
class IFdStreambuf: public std::streambuf
{
    protected:
        int d_fd;
        char d_buffer[1];
    public:
        IFdStreambuf(int fd);
    private:
        int underflow();
```
• The constructor initializes the buffer. However, the initialization sets `gptr`'s return value equal to `egptr`'s return value. This implies that the buffer is empty so `underflow` is immediately called to fill the buffer:

```cpp
inline IFdStreambuf::IFdStreambuf(int fd)
{
    d_fd(fd)
    setg(d_buffer, d_buffer + 1, d_buffer + 1);
}
```

• Finally `underflow` is overridden. The buffer is refilled by reading from the file descriptor. If this fails (for whatever reason), `EOF` is returned. More sophisticated implementations could act more intelligently here, of course. If the buffer could be refilled, `setg` is called to set up `streambuf`'s buffer pointers correctly:

```cpp
inline int IFdStreambuf::underflow()
{
    if (read(d_fd, d_buffer, 1) <= 0)
        return EOF;
    setg(d_buffer, d_buffer, d_buffer + 1);
    return *gptr();
}
```

The following `main` function shows how `IFdStreambuf` can be used:

```cpp
int main()
{
    IFdStreambuf fds(STDIN_FILENO);
    istream is(&fds);
    cout << is.rdbuf();
}
```

### 24.1.2.2 Using an n-character buffer

How complex would things get if we decided to use a buffer of substantial size? Not that complex. The following class allows us to specify the size of a buffer, but apart from that it is basically the same class as `IFdStreambuf` developed in the previous section. To make things a bit more interesting, in the class `IFdNStreambuf` developed here, the member `streambuf::xsgetn` is also overridden, to optimize reading a series of characters. Also a default constructor is provided that can be used in combination with the `open` member to construct an `istream` object before the file descriptor becomes available. In that case, once the descriptor becomes available, the `open` member can be used to initiate the object's buffer. Later, in section 24.2, we'll encounter such a situation.

To save some space, the success of various calls was not checked. In ‘real life’ implementations, these checks should of course not be omitted. The class `IFdNStreambuf` has the following characteristics:

• Its member functions use low-level functions operating on file descriptors. So apart from `streambuf` the `<unistd.h>` header file must have been read by the compiler before its member functions can be compiled.
24.1. USING FILE DESCRIPTORS WITH ‘STREAMBUF’ CLASSES

- As usual, it is derived from `std::streambuf`.

- Like the class `IFdStreambuf` (section 24.1.2.1), its data members are protected. Since the buffer's size is configurable, this size is kept in a dedicated data member, `d_bufsize`:

```cpp
class IFdNStreambuf: public std::streambuf
{
  protected:
    int     d_fd;
    size_t  d_bufsize;
    char*   d_buffer;
  public:
    IFdNStreambuf();
    IFdNStreambuf(int fd, size_t bufsize = 1);
    virtual ~IFdNStreambuf();
    void open(int fd, size_t bufsize = 1);
  private:
    virtual int underflow();
    virtual std::streamsize xsgetn(char *dest, std::streamsize n);
};
```

- The default constructor does not allocate a buffer. It can be used to construct an object before the file descriptor becomes known. A second constructor simply passes its arguments to `open`. `Open` will then initialize the object so that it can actually be used:

```cpp
inline IFdNStreambuf::IFdNStreambuf()
:
  d_bufsize(0),
  d_buffer(0)
{};
inline IFdNStreambuf::IFdNStreambuf(int fd, size_t bufsize)
{
  open(fd, bufsize);
}
```

- Once the object has been initialized by `open`, its destructor will both delete the object’s buffer and use the file descriptor to close the device:

```cpp
IFdNStreambuf::~IFdNStreambuf()
{
  if (d_bufsize)
  {
    close(d_fd);
    delete[] d_buffer;
  }
}
```

Even though the device is closed in the above implementation this may not always be desirable. In cases where the open file descriptor is already available the intention may be to use that descriptor repeatedly, each time using a newly constructed `IFdNStreambuf` object. It is left as an exercise to the reader to change this class in such a way that the device may optionally be closed. This approach was followed in, e.g., the Bobcat library\(^2\).

\(^2\)http://bobcat.sourceforge.net
The open member simply allocates the object's buffer. It is assumed that the calling program has already opened the device. Once the buffer has been allocated, the base class member setg is used to ensure that streambuf::eback, streambuf::gptr, and streambuf::egptr return correct values:

```cpp
void IFdNStreambuf::open(int fd, size_t bufsize)
{
    d_fd = fd;
    d_bufsize = bufsize;
    d_buffer = new char[d_bufsize];
    setg(d_buffer, d_buffer + d_bufsize, d_buffer + d_bufsize);
}
```

The overridden member underflow is implemented almost identically to IFdStreambuf’s (section 24.1.2.1) member. The only difference is that the current class supports buffers of larger sizes. Therefore, more characters (up to d_bufsize) may be read from the device at once:

```cpp
int IFdNStreambuf::underflow()
{
    if (gptr() < egptr())
        return *gptr();

    int nread = read(d_fd, d_buffer, d_bufsize);
    if (nread <= 0)
        return EOF;

    setg(d_buffer, d_buffer, d_buffer + nread);
    return *gptr();
}
```

Finally xsgetn is overridden. In a loop, n is reduced until 0, at which point the function terminates. Alternatively, the member returns if underflow fails to obtain more characters. This member optimizes the reading of series of characters. Instead of calling streambuf::sbumpc n times, a block of avail characters is copied to the destination, using streambuf::gbump to consume avail characters from the buffer using one function call:

```cpp
std::streamsize IFdNStreambuf::xsgetn(char *dest, std::streamsize n)
{
    int nread = 0;
    while (n)
    {
        if (!in_avail())
        {
            if (underflow() == EOF)
                break;
        }

        int avail = in_avail();
        if (avail > n)
            avail = n;
        ```
24.1. USING FILE DESCRIPTORS WITH 'STREAMBUF' CLASSES

```c
memcpy(dest + nread, gptr(), avail);
gbump(avail);

nread += avail;
n -= avail;
}

return nread;
}
```

The member function `xsgetn` is called by `streambuf::sgetn`, which is a streambuf member. Here is an example illustrating the use of this member function with an `IFdNStreambuf` object:

```c
#include <unistd.h>
#include <iostream>
#include <istream>
#include "ifdnbuf.h"
using namespace std;

int main()
{
    // internally: 30 char buffer
    IFdNStreambuf fds(STDIN_FILENO, 30);

    char buf[80]; // main() reads blocks of 80 chars
    while (true)
    {
        size_t n = fds.sgetn(buf, 80);
        if (n == 0)
            break;
        cout.write(buf, n);
    }
}
```

### 24.1.2.3 Seeking positions in 'streambuf' objects

When devices support seek operations, classes derived from `std::streambuf` should override the members `streambuf::seekoff` and `streambuf::seekpos`. The class `IFdSeek`, developed in this section, can be used to read information from devices supporting seek operations. The class `IFdSeek` was derived from `IFdStreambuf`, so it uses a character buffer of just one character. The facilities to perform seek operations, which are added to our new class `IFdSeek`, ensure that the input buffer is reset when a seek operation is requested. The class could also be derived from the class `IFdNStreambuf`. In that case the arguments to reset the input buffer must be adapted so that its second and third parameters point beyond the available input buffer. Let’s have a look at the characteristics of `IFdSeek`:

- As mentioned, `IFdSeek` is derived from `IFdStreambuf`. Like the latter class, `IFdSeek`'s member functions use facilities declared in `<unistd.h>`. So, the header file `<unistd.h>` must have been read by the compiler before it can compile the class’s members functions. To reduce the amount of typing when specifying types and constants from `streambuf` and `std::ios`, several typedefs are defined by the class. These typedefs refer to types that are defined in the header...
The class has a very basic interface. Its (only) constructor expects the device's file descriptor. It has no special tasks to perform and just calls its base class constructor:

```cpp
inline IFdSeek::IFdSeek(int fd) :
    IFdStreambuf(fd)
{}
```

The member `seek_off` is responsible for performing the actual seek operations. It calls `lseek` to seek a new position in a device whose file descriptor is known. If seeking succeeds, `setg` is called to define an already empty buffer, so that the base class's `underflow` member refills the buffer at the next input request.

```cpp
IFdSeek::pos_type IFdSeek::seekoff(off_type off, seekdir dir, openmode)
{
    pos_type pos =
    lseek
    (d_fd, off,
     (dir == std::ios::beg) ? SEEK_SET :
     (dir == std::ios::cur) ? SEEK_CUR :
                SEEK_END
    );
    if (pos < 0)
        return -1;
    setg(d_buffer, d_buffer + 1, d_buffer + 1);
    return pos;
}
```

Finally, the companion function `seekpos` is overridden as well: it is actually defined as a call to `seekoff`:

```cpp
inline IFdSeek::pos_type IFdSeek::seekpos(pos_type off, openmode mode)
{
    return seekoff(off, std::ios::beg, mode);
}
```
Here is an example of a program using the class IFdSeek. If this program is given its own source file using input redirection then seeking is supported (and with the exception of the first line, every other line is shown twice):

```cpp
#include "fdinseek.h"
#include <string>
#include <iostream>
#include <istream>
#include <iomanip>
using namespace std;

int main()
{
  IFdSeek fds(0);
  istream is(&fds);
  string s;

  while (true)
  {
    if (!getline(is, s))
      break;

    streampos pos = is.tellg();
    cout << setw(5) << pos << " ": " << s << ":\n";

    if (!getline(is, s))
      break;

    streampos pos2 = is.tellg();
    cout << setw(5) << pos2 << ": ": " << s << ":\n";

    if (!is.seekg(pos))
    {
      cout << "Seek failed\n";
      break;
    }
  }
}
```

### 24.1.2.4 Multiple ‘unget’ calls in ‘streambuf’ objects

Streambuf classes and classes derived from streambuf should support at least ungetting the last read character. Special care must be taken when series of unget calls must be supported. In this section the construction of a class supporting a configurable number of istream::unget or istream::putback calls is discussed.

Support for multiple (say ‘n’) unget calls is implemented by reserving an initial section of the input buffer, which is gradually filled up to contain the last n characters read. The class is implemented as follows:

- Once again, the class is derived from std::streambuf. It defines several data members,
allowing the class to perform the bookkeeping required to maintain an unget-buffer of a configurable size:

```cpp
class FdUnget: public std::streambuf
{
    int d_fd;
    size_t d_bufsize;
    size_t d_reserved;
    char *d_buffer;
    char *d_base;

public:
    FdUnget(int fd, size_t bufsz, size_t unget);
    virtual ~FdUnget();

private:
    int underflow();
};
```

- The class’s constructor expects a file descriptor, a buffer size and the number of characters that can be ungot or pushed back as its arguments. This number determines the size of a reserved area, defined as the first d_reserved bytes of the class’s input buffer.

  - The input buffer will always be at least one byte larger than d_reserved. So, a certain number of bytes may be read. Once d_reserved bytes have been read at most d_reserved bytes can be ungot.

  - Next, the starting point for reading operations is configured. It is called d_base, pointing to a location d_reserved bytes beyond the location represented by d_buffer. This is always the location where buffer refills start.

  - Now that the buffer has been constructed, we’re ready to define streambuf’s buffer pointers using setg. As no characters have been read yet, all pointers are set to point to d_base. If unget is called at this point, no characters are available, and unget (correctly) fails.

  - Eventually, the refill buffer’s size is determined as the number of allocated bytes minus the size of the reserved area.

Here is the class’s constructor:

```cpp
FdUnget::FdUnget(int fd, size_t bufsz, size_t unget)
    :
    d_fd(fd),
    d_reserved(unget)
{
    size_t allocate =
        bufsz > d_reserved ?
        bufsz
    :
        d_reserved + 1;

    d_buffer = new char[allocate];
    d_base = d_buffer + d_reserved;
    setg(d_base, d_base, d_base);
    d_bufsize = allocate - d_reserved;
}
```
• The class’s destructor simply returns the memory allocated for the buffer to the common pool:

```cpp
inline FdUnget::~FdUnget()
{
    delete[] d_buffer;
}
```

• Finally, underflow is overridden as follows:

  – First underflow determines the number of characters that could potentially be ungot. If that number of characters are ungot, the input buffer is exhausted. So this value may be any value between 0 (the initial state) or the input buffer’s size (when the reserved area has been filled up completely, and all current characters in the remaining section of the buffer have also been read);

  – Next the number of bytes to move into the reserved area is computed. This number is at most d_reserved, but it is set equal to the actual number of characters that can be ungot if this value is smaller;

  – Now that the number of characters to move into the reserved area is known, this number of characters is moved from the input buffer’s end to the area immediately before d_base;

  – Then the buffer is refilled. This all is standard, but notice that reading starts from d_base and not from d_buffer;

  – Finally, streambuf’s read buffer pointers are set up. Eback is set to move locations before d_base, thus defining the guaranteed unget-area, gptr is set to d_base, since that’s the location of the first read character after a refill, and egptr is set just beyond the location of the last character read into the buffer.

Here is underflow’s implementation:

```cpp
int FdUnget::underflow()
{
    size_t ungetsize = gptr() - eback();
    size_t move = std::min(ungetsize, d_reserved);
    memcpy(d_base - move, egptr() - move, move);

    int nread = read(d_fd, d_base, d_bufsize);
    if (nread <= 0) // none read -> return EOF
        return EOF;
    setg(d_base - move, d_base, d_base + nread);
    return *gptr();
}
```

**An example using FdUnget**

The next example program illustrates the use of the class FdUnget. It reads at most 10 characters from the standard input, stopping at EOF. A guaranteed unget-buffer of 2 characters is defined in a buffer holding 3 characters. Just before reading a character, the program tries to unget at most 6 characters. This is, of course, not possible; but the program nicely ungets as many characters as possible, considering the actual number of characters read:

```cpp
#include "fdunget.h"
#include <string>
```
#include <iostream>
#include <istream>
using namespace std;

int main()
{
    FdUnget fds(0, 3, 2);
    istream is(&fds);
    char c;

    for (int idx = 0; idx < 10; ++idx)
    {
        cout << "after reading " << idx << " characters:\n";
        for (int ug = 0; ug <= 6; ++ug)
        {
            if (!is.unget())
            {
                cout << "unget failed at attempt " << (ug + 1) << "\n" << "rereading: ";
                is.clear();
                while (ug--)
                {
                    is.get(c);
                    cout << c;
                }
                cout << "\n";
                break;
            }
        }
        if (!is.get(c))
        {
            cout << " reached\n";
            break;
        }
        cout << "Next character: " << c << \n';
    }
/*
Generated output after 'echo abcde | program':
after reading 0 characters:
    unget failed at attempt 1
    rereading: ''
Next character: a
after reading 1 characters:
    unget failed at attempt 2
    rereading: 'a'
Next character: b
after reading 2 characters:
    unget failed at attempt 3
    rereading: 'ab'
*/
24.1.3 Fixed-sized field extraction from istream objects

Usually when extracting information from istream objects `operator>>`, the standard extraction operator is perfectly suited for the task as in most cases the extracted fields are white-space (or otherwise clearly) separated from each other. But this does not hold true in all situations. For example, when a web-form is posted to some processing script or program, the receiving program may receive the form field's values as url-encoded characters: letters and digits are sent unaltered, blanks are sent as + characters, and all other characters start with % followed by the character's ascii-value represented by its two digit hexadecimal value.

When decoding url-encoded information, simple hexadecimal extraction won't work, as that extracts as many hexadecimal characters as available, instead of just two. Since the letters a-f ' and 0-9 are legal hexadecimal characters, a text like `My name is 'Ed` results in the extraction of the hexadecimal values 60ed and 27, instead of 60 and 27. The name Ed disappears from view, which is clearly not what we want.

In this case, having seen the %, we could extract 2 characters, put them in an istream object, and extract the hexadecimal value from the istream object. A bit cumbersome, but doable. Other approaches are possible as well.

The class Fistream for fixed-sized field istream defines an istream class supporting both fixed-sized field extractions and blank-delimited extractions (as well as unformatted read calls). The class may be initialized as a wrapper around an existing istream, or it can be initialized using the name of an existing file. The class is derived from istream, allowing all extractions and operations supported by istreams in general. Fistream defines the following data members:

- `d_filebuf`: a filebuffer used when Fistream reads its information from a named (existing) file. Since the filebuffer is only needed in that case, and since it must be allocated dynamically, it is defined as a `unique_ptr<filebuf>` object.
CHAPTER 24. CONCRETE EXAMPLES

- d_streambuf: a pointer to Fistream's streambuf. It points to d_filebuf when Fistream opens a file by name. When an existing istream is used to construct an Fistream, it points to the existing istream's streambuf.

- d_iss: an istringstream object used for the fixed field extractions.

- d_width: a size_t indicating the width of the field to extract. If 0 no fixed field extractions is used, but information is extracted from the istream base class object using standard extractions.

Here is the initial section of Fistream's class interface:

```cpp
class Fistream: public std::istream
{
    std::unique_ptr<std::filebuf> d_filebuf;
    std::streambuf *d_streambuf;
    std::istringstream d_iss;
    size_t d_width;
}
```

As stated, Fistream objects can be constructed from either a filename or an existing istream object. The class interface therefore declares two constructors:

```cpp
Fistream(std::istream &stream);
Fistream(char const *name, std::ios::openmode mode = std::ios::in);
```

When an Fistream object is constructed using an existing istream object, the Fistream's istream part simply uses the stream's streambuf object:

```cpp
Fistream::Fistream(istream &stream)
:
    istream(stream.rdbuf()),
    d_streambuf(rdbuf()),
    d_width(0)
{}
```

When an ofstream object is constructed using a filename, the istream base initializer is given a new filebuf object to be used as its streambuf. Since the class's data members are not initialized before the class's base class has been constructed, d_filebuf can only be initialized thereafter. By then, the filebuf is only available as rdbuf, returning a streambuf. However, as it is actually a filebuf, a static_cast is used to cast the streambuf pointer returned by rdbuf to a filebuf *, so d_filebuf can be initialized:

```cpp
Fistream::Fistream(char const *name, ios::openmode mode)
:
    istream(new filebuf()),
    d_filebuf(static_cast<filebuf*>(rdbuf())),
    d_streambuf(d_filebuf.get()),
    d_width(0)
{
    d_filebuf->open(name, mode);
}
24.1. USING FILE DESCRIPTORS WITH ‘STREAMBUF’ CLASSES

24.1.3.1 Member functions and example

There is only one additional public member: `setField(field const &)`. This member defines the size of the next field to extract. Its parameter is a reference to a `field` class, a `manipulator class` defining the width of the next field.

Since a `field` is mentioned in `Fistream`'s interface, `field` must be declared before `Fistream`'s interface starts. The class `field` itself is simple and declares `Fistream` as its friend. It has two data members: `d_width` specifies the width of the next field, and `d_newWidth` which is set to true if `d_width`'s value should actually be used. If `d_newWidth` is false, `Fistream` returns to its standard extraction mode. The class `field` has two constructors: a default constructor, setting `d_newWidth` to false, and a second constructor expecting the width of the next field to extract as its value. Here is the class `field`:

```cpp
class field
{
    friend class Fistream;
    size_t d_width;
    bool    d_newWidth;

    public:
        field(size_t width);
        field();
};
inline field::field(size_t width)
:
    d_width(width),
    d_newWidth(true)
{ }
inline field::field()
:
    d_newWidth(false)
{ }
```

Since `field` declares `Fistream` as its friend, `setField` may inspect `field`'s members directly.

Time to return to `setField`. This function expects a reference to a `field` object, initialized in one of three different ways:

- **field()**: When `setField`'s argument is a `field` object constructed by its default constructor the next extraction will use the same fieldwidth as the previous extraction.

- **field(0)**: When this `field` object is used as `setField`'s argument, fixed-sized field extraction stops, and the `Fistream` acts like any standard `istream` object again.

- **field(x)**: When the `field` object itself is initialized by a non-zero `size_t` value `x`, then the next field width is `x` characters wide. The preparation of such a field is left to `setBuffer`, `Fistream`'s only private member.

Here is `setField`'s implementation:

```cpp
std::istream &Fistream::setField(field const &params)
```
{ 
    if (params.d_newWidth) // new field size requested
        d_width = params.d_width; // set new width
    if (!d_width) // no width?
        rdbuf(d_streambuf); // return to the old buffer
    else
        setBuffer(); // define the extraction buffer

    return *this;
}

The private member setBuffer defines a buffer of \(d\_width + 1\) characters and uses read to fill the buffer with \(d\_width\) characters. The buffer is an NTBS. This buffer is used to initialize the \(d\_iss\) member. Fistream’s rdbuf member is used to extract the \(d\_str\)’s data via the Fistream object itself:

void Fistream::setBuffer()
{
    char *buffer = new char[d_width + 1]; // use istream’s buffer to
    rdbuf(d_streambuf);
    buffer[read(buffer, d_width).gcount()] = 0; // read d_width chars,
    // terminated by a 0-byte
    d_iss.str(buffer);
    delete[] buffer;

    rdbuf(d_iss.rdbuf()); // switch buffers
}

Although setField could be used to configure Fistream to use or not to use fixed-sized field extraction, using manipulators is probably preferable. To allow field objects to be used as manipulators an overloaded extraction operator was defined. This extraction operator accepts istream & and a field const & objects. Using this extraction operator, statements like

\[\text{fis} \gg \text{field(2)} \gg x \gg \text{field(0)};\]

are possible (assuming fis is a Fistream object). Here is the overloaded operator>>, as well as its declaration:

istream &std::operator>>(istream &str, field const &params)
{
    return static_cast<Fistream *>(str)->setField(params);
}

Declaration:

amespace std
{
    istream &operator>>(istream &str, FBB::field const &params);
}
24.2. THE ‘FORK’ SYSTEM CALL

Finally, an example. The following program uses a Fistream object to url-decode url-encoded information appearing at its standard input:

```cpp
int main()
{
    Fistream fis(cin);
    fis >> hex;
    while (true)
    {
        size_t x;
        switch (x = fis.get())
        {
        case '\n':
            cout << '\n';
            break;
        case '+':
            cout << ' ';
            break;
        case '%':
            fis >> field(2) >> x >> field(0);
            // FALLING THROUGH
        default:
            cout << static_cast<char>(x);
            break;
        case EOF:
            return 0;
        }
    }
    /*
    Generated output after:
    echo My+name+is+%60Ed%27 | a.out
    My name is ‘Ed’
    */
}
```

24.2 The ‘fork’ system call

From the C programming language the fork system call is well known. When a program needs to start a new process, system can be used. The function system requires the program to wait for the child process to terminate. The more general way to spawn subprocesses is to use fork.

In this section we investigate how C++ can be used to wrap classes around a complex system call like fork. Much of what follows in this section directly applies to the Unix operating system, and the discussion therefore focuses on that operating system. Other systems usually provide comparable facilities. What follows is closely related to the Template Design Pattern (cf. Gamma et al. (1995) Design Patterns, Addison-Wesley)

When fork is called, the current program is duplicated in memory, thus creating a new process. Following this duplication both processes continue their execution just below the fork system call. The two processes may inspect fork’s return value: the return value in the original process (called the parent process) differs from the return value in the newly created process (called the child process):
• In the parent process fork returns the process ID of the (child) process that was created by the fork system call. This is a positive integer value.

• In the child process fork returns 0.

• If fork fails, -1 is returned.

24.2.1 A basic Fork class

A basic Fork class should hide all bookkeeping details of a system call like fork from its users. The class Fork developed here does just that. The class itself only ensures the proper execution of the fork system call. Normally, fork is called to start a child process, usually boiling down to the execution of a separate process. This child process may expect input at its standard input stream and/or may generate output to its standard output and/or standard error streams. Fork does not know all this, and does not have to know what the child process will do. Fork objects should be able to start their child processes.

Fork’s constructor cannot know what actions its child process should perform. Similarly, it cannot know what actions the parent process should perform. For these kind of situations, the template method design pattern was developed. According to Gamma c.s., the template method design pattern "Define(s) the skeleton of an algorithm in an operation, deferring some steps to sub-classes. [The] Template Method (design pattern) lets subclasses redefine certain steps of an algorithm, without changing the algorithm’s structure."

This design pattern allows us to define an abstract base class already providing the essential steps related to the fork system call, deferring the implementation of other parts of the fork system call to subclasses.

The Fork abstract base class has the following characteristics:

• It defines a data member d_pid. In the parent process this data member contains the child’s process id and in the child process it has the value 0. Its public interface declares only two members:
  - a fork member function, responsible for the actual forking (i.e., it creates the (new) child process);
  - a virtual destructor ~Fork (having an empty body).

Here is Fork’s interface:

```cpp
class Fork {
  int d_pid;

public:
  virtual ~Fork();
  void fork();

protected:
  int pid() const;
  int waitForChild(); // returns the status

private:
```


virtual void childRedirections();
virtual void parentRedirections();

virtual void childProcess() = 0; // pure virtual members
virtual void parentProcess() = 0;
);

• All other non-virtual member functions are declared in the class's protected section and can thus only be used by derived classes. They are:

  - pid(): The member function pid allows derived classes to access the system fork's return value:

    inline int Fork::pid() const
    {
        return d_pid;
    }

  - waitForChild(): The member int waitForChild can be called by parent processes to wait for the completion of their child processes (as discussed below). This member is declared in the class interface. Its implementation is:

    #include "fork.ih"

    int Fork::waitForChild()
    {
        int status;

        waitpid(d_pid, &status, 0);

        return WEXITSTATUS(status);
    }

    This simple implementation returns the child's exit status to the parent. The called system function waitpid blocks until the child terminates.

• When fork system calls are used, parent processes and child processes must always be distinguished. The main distinction between these processes is that d_pid becomes the child's process-id in the parent process, while d_pid becomes 0 in the child process itself. Since these two processes must always be distinguished (and present), their implementation by classes derived from Fork is enforced by Fork's interface: the members childProcess and parentProcess, defining the child process' actions and parentProcess, defining the parent process' actions were defined as pure virtual functions.

• communication between parent- and child processes may use standard streams or other facilities, like pipes (cf. section 24.2.5). To facilitate this inter-process communication, derived classes may implement:

  - childRedirections(): this member should be overridden by derived classes if any standard stream (cin, cout,) or cerr must be redirected in the child process (cf. section 24.2.3). By default it has an empty implementation;

  - parentRedirections(): this member should be overridden by derived classes if any standard stream (cin, cout,) or cerr must be redirected in the parent process. By default it has an empty implementation.

Redirection of the standard streams is necessary if parent and child processes must communicate with each other via the standard streams. Here are their default definitions. Since these
functions are virtual functions they should not be implemented inline, but in their own source file:

```cpp
void Fork::childRedirections()
{
}
void Fork::parentRedirections()
{
}
```

## 24.2.2 Parents and Children

The member function `fork` calls the system function `fork` (Caution: since the system function `fork` is called by a member function having the same name, the `::` scope resolution operator must be used to prevent a recursive call of the member function itself). The function `::fork`'s return value determines whether `parentProcess` or `childProcess` is called. Maybe redirection is necessary. `Fork::fork`'s implementation calls `childRedirections` just before calling `childProcess`, and `parentRedirections` just before calling `parentProcess`:

```cpp
#include "fork.ih"
void Fork::fork()
{
    if ((d_pid = ::fork()) < 0)
        throw "Fork::fork() failed";
    if (d_pid == 0) // childprocess has pid == 0
    {
        childRedirections();
        childProcess();
        exit(1); // we shouldn’t come here:
    } // childProcess() should exit
    parentRedirections();
    parentProcess();
}
```

In `fork.cc` the class’s *internal header file* `fork.ih` is included. This header file takes care of the inclusion of the necessary system header files, as well as the inclusion of `fork.h` itself. Its implementation is:

```cpp
#include "fork.h"
#include <cstdlib>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

Child processes should not return: once they have completed their tasks, they should terminate. This happens automatically when the child process performs a call to a member of the `exec` family, but if the child itself remains active, then it must make sure that it terminates properly. A child process normally uses `exit` to terminate itself, but note that `exit` prevents the activation of destructors of objects defined at the same or more superficial nesting levels than the level at which `exit` is called. Destructors of globally defined objects are activated when `exit` is used. When using `exit` to terminate `childProcess`, it should either itself call a support member function defining
all nested objects it needs, or it should define all its objects in a compound statement (e.g., using a throw block) calling exit beyond the compound statement.

Parent processes should normally wait for their children to complete. Terminating child processes inform their parents that they are about to terminate by sending a signal that should be caught by their parents. If child processes terminate and their parent processes do not catch those signals then such child processes remain visible as so-called zombie processes.

If parent processes must wait for their children to complete, they may call the member waitForChild. This member returns the exit status of a child process to its parent.

There exists a situation where the child process continues to live, but the parent dies. This is a fairly natural event: parents tend to die before their children do. In our context (i.e. C++), this is called a daemon program. In a daemon the parent process dies and the child program continues to run as a child of the basic init process. Again, when the child eventually dies a signal is sent to its ‘step-parent’ init. This does not create a zombie as init catches the termination signals of all its (step-) children. The construction of a daemon process is very simple, given the availability of the class Fork (cf. section 24.2.4).

### 24.2.3 Redirection revisited

Earlier, in section 6.6.1 streams were redirected using the ios::rdbuf member function. By assigning the streambuf of a stream to another stream, both stream objects access the same streambuf, thus implementing redirection at the level of the programming language itself.

This may be fine within the context of a C++ program, but once we leave that context the redirection terminates. The operating system does not know about streambuf objects. This situation is encountered, e.g., when a program uses a system call to start a subprogram. The example program at the end of this section uses C++ redirection to redirect the information inserted into cout to a file, and then calls

```cpp
system("echo hello world")
```

to echo a well-known line of text. Since echo writes its information to the standard output, this would be the program’s redirected file if the operating system would recognize C++’s redirection.

But redirection doesn’t happen. Instead, hello world still appears at the program’s standard output and the redirected file is left untouched. To write hello world to the redirected file redirection must be realized at the operating system level. Some operating systems (e.g., Unix and friends) provide system calls like dup and dup2 to accomplish this. Examples of the use of these system calls are given in section 24.2.5.

Here is the example of the failing redirection at the system level following C++ redirection using streambuf redirection:

```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

int main()
{
    ofstream of("outfile");
```
Applications exist in which the only purpose of fork is to start a child process. The parent process terminates immediately after spawning the child process. If this happens, the child process continues to run as a child process of init, the always running first process on Unix systems. Such a process is often called a daemon, running as a background process.

Although the next example can easily be constructed as a plain C program, it was included in the C++ Annotations because it is so closely related to the current discussion of the Fork class. I thought about adding a daemon member to that class, but eventually decided against it because the construction of a daemon program is very simple and requires no features other than those currently offered by the class Fork. Here is an example illustrating the construction of such a daemon program. Its child process doesn't do exit but throw 0 which is caught by the catch clause of the child's main function. Doing this ensures that any objects defined by the child process are properly destroyed:

```cpp
#include <iostream>
#include <unistd.h>
#include "fork.h"

class Daemon: public Fork
{
    virtual void parentProcess() // the parent does nothing.
    {}

    virtual void childProcess() // actions by the child
    
    {sleep(3);  // just a message...
      std::cout << "Hello from the child process\n";
      throw 0;  // The child process ends
    }
};

int main()
try
{
    Daemon().fork();
}
```

24.2.4 The 'Daemon' program
24.2. THE ‘FORK’ SYSTEM CALL

24.2.5 The class ‘Pipe’

Redirection at the system level requires the use of file descriptors, created by the pipe system call. When two processes want to communicate using such file descriptors, the following happens:

- The process constructs two associated file descriptors using the pipe system call. One of the file descriptors is used for writing, the other file descriptor is used for reading.
- Forking takes place (i.e., the system fork function is called), duplicating the file descriptors. Now we have four file descriptors as the child process and the parent process both have their own copies of the two file descriptors created by pipe.
- One process (say, the parent process) uses the file descriptors for reading. It should close its file descriptor intended for writing.
- The other process (say, the child process) uses the file descriptors for writing. It should therefore close its file descriptor intended for reading.
- All information written by the child process to the file descriptor intended for writing, can now be read by the parent process from the corresponding file descriptor intended for reading, thus establishing a communication channel between the child and the parent process.

Though basically simple, errors may easily creep in. Functions of file descriptors available to the two processes (child or parent) may easily get mixed up. To prevent bookkeeping errors, the bookkeeping may be properly set up once, to be hidden thereafter inside a class like the Pipe class developed here. Let’s have a look at its characteristics (before using functions like pipe and dup the compiler must have read the <unistd.h> header file):

- The pipe system call expects a pointer to two int values, representing, respectively, the file descriptor used for reading and the file descriptor used for writing. To avoid confusion, the class Pipe defines an enum having values associating the indices of the array of 2 ints with symbolic constants. The two file descriptors themselves are stored in a data member d_fd. Here is the initial section of the class’s interface:

```cpp
class Pipe
{
    enum RW { READ, WRITE };
    int d_fd[2];
```

- The class only needs a default constructor. This constructor calls pipe to create a set of associated file descriptors used for accessing both ends of a pipe:

```cpp
Pipe::Pipe()
{
```
if (pipe(d_fd))
    throw "Pipe::Pipe(): pipe() failed";
}

- The members `readOnly` and `readFrom` are used to configure the pipe's reading end. The latter function is used when using redirection. It is provided with an alternate file descriptor to be used for reading from the pipe. Usually this alternate file descriptor is `STDIN_FILENO`, allowing `cin` to extract information from the pipe. The former function is merely used to configure the reading end of the pipe. It closes the matching writing end and returns a file descriptor that can be used to read from the pipe:

```cpp
text
int Pipe::readOnly()
{
    close(d_fd[WRITE]);
    return d_fd[READ];
}
void Pipe::readFrom(int fd)
{
    readOnly();

    redirect(d_fd[READ], fd);
    close(d_fd[READ]);
}
```

- `writeOnly` and two `writtenBy` members are available to configure the writing end of a pipe. The former function is only used to configure the writing end of the pipe. It closes the reading end, and returns a file descriptor that can be used for writing to the pipe:

```cpp
text
int Pipe::writeOnly()
{
    close(d_fd[READ]);
    return d_fd[WRITE];
}
void Pipe::writtenBy(int fd)
{
    writtenBy(&fd, 1);
}
void Pipe::writtenBy(int const *fd, size_t n)
{
    writeOnly();

    for (size_t idx = 0; idx < n; idx++)
        redirect(d_fd[WRITE], fd[idx]);

    close(d_fd[WRITE]);
}
```

For the latter member two overloaded versions are available:
- `writtenBy(int fd)` is used to configure single redirection, so that a specific file descriptor (usually `STDOUT_FILENO` or `STDERR_FILENO`) can be used to write to the pipe;
- `(writtenBy(int const *fd, size_t n))` may be used to configure multiple redirection, providing an array argument containing file descriptors. Information written to any of these file descriptors is actually written to the pipe.
24.2. THE 'FORK' SYSTEM CALL

- The class has one private data member, redirect, used to set up redirection through the dup2 system call. This function expects two file descriptors. The first file descriptor represents a file descriptor that can be used to access the device's information; the second file descriptor is an alternate file descriptor that may also be used to access the device's information. Here is redirect's implementation:

```c
void Pipe::redirect(int d_fd, int alternateFd)
{
    if (dup2(d_fd, alternateFd) < 0)
        throw "Pipe: redirection failed";
}
```

Now that redirection can be configured easily using one or more Pipe objects, we'll use Fork and Pipe in various example programs.

24.2.6 The class 'ParentSlurp'

The class ParentSlurp, derived from Fork, starts a child process executing a stand-alone program (like /bin/ls). The (standard) output of the executed program is not shown on the screen but is read by the parent process.

For demonstration purposes the parent process writes the lines it receives to its standard output stream, prepending linenumbers to the lines. It is attractive to redirect the parent's standard input stream to allow the parent to read the output from the child process using its std::cin input stream. Therefore, the only pipe in the program is used as an input pipe for the parent, and an output pipe for the child.

The class ParentSlurp has the following characteristics:

- It is derived from Fork. Before starting ParentSlurp's class interface, the compiler must have read fork.h and pipe.h. The class only uses one data member, a Pipe object d_pipe.

- As Pipe's constructor already defines a pipe, and as d_pipe is automatically initialized by ParentSlurp's default constructor, which is implicitly provided, all additional members only exist for ParentSlurp's own benefit so they can be defined in the class's (implicit) private section. Here is the class's interface:

```c
class ParentSlurp: public Fork
{
    Pipe     d_pipe;
    virtual void childRedirections();
    virtual void parentRedirections();
    virtual void childProcess();
    virtual void parentProcess();
};
```

- The childRedirections member configures the writing end of the pipe. So, all information written to the child's standard output stream ends up in the pipe. The big advantage of this is that no additional streams are needed to write to a file descriptor:

```c
inline void ParentSlurp::childRedirections()
{
    d_pipe.writtenBy(STDOUT_FILENO);
}
```
• The `parentRedirections` member, configures the reading end of the pipe. It does so by connecting the reading end of the pipe to the parent's standard input file descriptor (`STDIN_FILENO`). This allows the parent to perform extractions from `cin`, not requiring any additional streams for reading.

```cpp
inline void ParentSlurp::parentRedirections()
{
    d_pipe.readFrom(STDIN_FILENO);
}
```

• The `childProcess` member only needs to concentrate on its own actions. As it only needs to execute a program (writing information to its standard output), the member can consist of one single statement:

```cpp
inline void ParentSlurp::childProcess()
{
    execl("/bin/ls", "/bin/ls", 0);
}
```

• The `parentProcess` member simply 'slurps' the information appearing at its standard input. Doing so, it actually reads the child's output. It copies the received lines to its standard output stream prefixing line numbers to them:

```cpp
void ParentSlurp::parentProcess()
{
    std::string line;
    size_t nr = 1;

    while (getline(std::cin, line))
        std::cout << nr++ << " : " << line << '\n';

    waitForChild();
}
```

The following program simply constructs a `ParentSlurp` object, and calls its `fork()` member. Its output consists of a numbered list of files in the directory where the program is started. Note that the program also needs the `fork.o`, `pipe.o` and `waitforchild.o` object files (see earlier sources):

```cpp
int main()
{
    ParentSlurp().fork();
}
/*
 Generated Output (example only, actually obtained output may differ):
 1: a.out
 2: bitand.h
 3: bitfunctional
 4: bitnot.h
 5: daemon.cc
 6: fdinseek.cc
 7: fdinseek.h
 ... */
```
24.2. THE ‘FORK’ SYSTEM CALL

24.2.7 Communicating with multiple children

The next step up the ladder is the construction of a child-process monitor. Here, the parent process is responsible for all its child processes, but it also must read their standard output. The user enters information at the standard input of the parent process. A simple command language is used for this:

- **start**: this start a new child process. The parent returns the child’s ID (a number) to the user. The ID is thereupon be used to identify a particular child process
- **<nr> text** sends “text” to the child process having ID <nr>;
- **stop <nr>** terminates the child process having ID <nr>;
- **exit** terminates the parent as well as all its child processes.

If a child process hasn’t received text for some time it will complain by sending a message to the parent-process. Those messages are simply transmitted to the user by copying them to the standard output stream.

A problem with programs like our monitor is that they allow asynchronous input from multiple sources. Input may appear at the standard input as well as at the input-sides of pipes. Also, multiple output channels are used. To handle situations like these, the **select** system call was developed.

### 24.2.7.1 The class ‘Selector’: interface

The **select** system call was developed to handle asynchronous **I/O multiplexing**. The **select** system call is used to handle, e.g., input appearing simultaneously at a set of file descriptors.

The **select** function is rather complex, and its full discussion is beyond the C++ Annotations’ scope. By encapsulating **select** in a class **Selector**, hiding its details and offering an intuitively attractive interface, its use is simplified. The **Selector** class has these features:

- **Efficiency.** As most of **Selector**’s members are very small, most members can be implemented inline. The class requires quite a few data members. Most of these data members belong to types that require some system headers to be included first:

  ```
  #include <limits.h>
  #include <unistd.h>
  #include <sys/time.h>
  #include <sys/types.h>
  ```

  - The class interface can now be defined. The data type **fd_set** is a type designed to be used by **select** and variables of this type contain the set of file descriptors on which **select** may sense some activity. Furthermore, **select** allows us to fire an **asynchronous alarm**. To set the alarm time, the class **Selector** defines a **timeval** data member. Other members are used for internal bookkeeping purposes. Here is the **class Selector**'s interface:

    ```
    class Selector
    {
        fd_set d_read;
        fd_set d_write;
        fd_set d_except;
        fd_set d_ret_read;
    ```
24.2.7.2 The class ‘Selector’: implementation

Selector’s member functions serve the following tasks:

- **Selector():** the (default) constructor. It clears the read, write, and execute fd_set variables, and switches off the alarm. Except for d_max, the remaining data members do not require specific initializations:

  ```cpp
  Selector::Selector()
  {
      FD_ZERO(&d_read);
      FD_ZERO(&d_write);
      FD_ZERO(&d_except);
      noAlarm();
      d_max = 0;
  }
  ```

- **int wait():** this member blocks until the alarm times out or until activity is sensed at any of the file descriptors monitored by the Selector object. It throws an exception when the select system call itself fails:

  ```cpp
  int Selector::wait()
  ```
24.2. THE 'FORK' SYSTEM CALL

```
{    timeval t = d_alarm;
    d_ret_read = d_read;
    d_ret_write = d_write;
    d_ret_except = d_except;

    d_readidx = 0;
    d_writeidx = 0;
    d_exceptidx = 0;

    d_ret = select(d_max, &d_ret_read, &d_ret_write, &d_ret_except,
                   t.tv_sec == -1 && t.tv_usec == -1 ? 0 : &t);

    if (d_ret < 0)
        throw "Selector::wait()/select() failed";

    return d_ret;
}
```

- int nReady: this member function’s return value is only defined when `wait` has returned. In that case it returns 0 for an alarm-timeout, -1 if `select` failed, and otherwise the number of file descriptors on which activity was sensed:

  ```cpp```
  inline int Selector::nReady()
  {
    return d_ret;
  }
  ```cpp```

- int readFd(): this member function’s return value is also only defined after `wait` has returned. Its return value is -1 if no (more) input file descriptors are available. Otherwise the next file descriptor available for reading is returned:

  ```cpp```
  inline int Selector::readFd()
  {
    return checkSet(&d_readidx, d_ret_read);
  }
  ```cpp```

- int writeFd(): operating analogously to `readFd`, it returns the next file descriptor to which output is written. It uses `d_writeidx` and `d_ret_read` and is implemented analogously to `readFd`;

- int exceptFd(): operating analogously to `readFd`, it returns the next exception file descriptor on which activity was sensed. It uses `d_except_idx` and `d_ret_except` and is implemented analogously to `readFd`;

- void setAlarm(int sec, int usec = 0): this member activates Select’s alarm facility. At least the number of seconds to wait for the alarm to go off must be specified. It simply assigns values to `d_alarm`’s fields. At the next Select::wait call, the alarm fires (i.e., `wait` returns with return value 0) once the configured alarm-interval has passed:

  ```cpp```
  inline void Selector::setAlarm(int sec, int usec)
  {
    d_alarm.tv_sec  = sec;
    d_alarm.tv_usec = usec;
  }
  ```cpp```
• void noAlarm(): this member switches off the alarm, by simply setting the alarm interval to a very long period:

```cpp
inline void Selector::noAlarm()
{
    setAlarm(-1, -1);
}
```

• void addReadFd(int fd): this member adds a file descriptor to the set of input file descriptors monitored by the Selector object. The member function wait returns once input is available at the indicated file descriptor:

```cpp
inline void Selector::addReadFd(int fd)
{
    addFd(&d_read, fd);
}
```

• void addWriteFd(int fd): this member adds a file descriptor to the set of output file descriptors monitored by the Selector object. The member function wait returns once output is available at the indicated file descriptor. Using d_write, it is implemented analogously to addReadFd;

• void addExceptFd(int fd): this member adds a file descriptor to the set of exception file descriptors to be monitored by the Selector object. The member function wait returns once activity is sensed at the indicated file descriptor. Using d_except, it is implemented analogously to addReadFd;

• void rmReadFd(int fd): this member removes a file descriptor from the set of input file descriptors monitored by the Selector object:

```cpp
inline void Selector::rmReadFd(int fd)
{
    FD_CLR(fd, &d_read);
}
```

• void rmWriteFd(int fd): this member removes a file descriptor from the set of output file descriptors monitored by the Selector object. Using d_write, it is implemented analogously to rmReadFd;

• void rmExceptFd(int fd): this member removes a file descriptor from the set of exception file descriptors to be monitored by the Selector object. Using d_except, it is implemented analogously to rmReadFd;

The class's remaining (two) members are support members, and should not be used by non-member functions. Therefore, they are declared in the class's private section:

• The member addFd adds a file descriptor to a fd_set:

```cpp
void Selector::addFd(fd_set *set, int fd)
{
    FD_SET(fd, set);
    if (fd >= d_max)
        d_max = fd + 1;
}
```
• The member checkSet tests whether a file descriptor (*index) is found in a fd_set:

```cpp
int Selector::checkSet(int *index, fd_set &set) {
    int &idx = *index;
    while (idx < d_max && !FD_ISSET(idx, &set))
        ++idx;
    return idx == d_max ? -1 : idx++;
}
```

### 24.2.7.3 The class ‘Monitor’: interface

The monitor program uses a Monitor object doing most of the work. The class Monitor’s public interface only offers a default constructor and one member, run, to perform its tasks. All other member functions are located in the class’s private section.

Monitor defines the private enum Commands, symbolically listing the various commands its input language supports, as well as several data members. Among the data members are a Selector object and a map using child order numbers as its keys and pointer to Child objects (see section 24.2.7.7) as its values. Furthermore, Monitor has a static array member s_handler[], storing pointers to member functions handling user commands.

A destructor should be implemented as well, but its implementation is left as an exercise to the reader. Here is Monitor’s interface, including the interface of the nested class Find that is used to create a function object:

```cpp
class Monitor {
    enum Commands {
        UNKNOWN, START, EXIT, STOP, TEXT,
        sizeofCommands
    };  
typedef std::map<int, std::shared_ptr<Child>> MapIntChild;

    friend class Find;
    class Find {
        int d_nr;
        public:
            Find(int nr);
            bool operator() (MapIntChild::value_type &vt) const;
    };

    Selector d_selector;
    int d_nr;
    MapIntChild d_child;
};
```
static void (Monitor::*s_handler[])(int, std::string const &);
static int s_initialize;

public:
    enum Done
    {};
    Monitor();
    void run();

private:
    static void killChild(MapIntChild::value_type it);
    static int initialize();
    Commands next(int *value, std::string *line);
    void processInput();
    void processChild(int fd);
    void createNewChild(int, std::string const &);
    void exiting(int = 0, std::string const &msg = std::string());
    void sendChild(int value, std::string const &);
    void stopChild(int value, std::string const &);
    void unknown(int, std::string const &);
};

Since there's only one non-class type data member, the class's constructor is a very simple function which could be implemented inline:

    inline Monitor::Monitor()
    :
        d_nr(0)
    {}

24.2.7.4 The class 'Monitor': s_handler

The array s_handler, storing pointers to functions needs to be initialized as well. This can be accomplished in several ways:

- Since the Commands enumeration only specifies a fairly limited set of commands, compile-time initialization could be considered:

    void (Monitor::*Monitor::s_handler[])(int, string const &) =
    {
        &Monitor::unknown, // order follows enum Command's
        &Monitor::createNewChild, // elements
        &Monitor::exiting,
        &Monitor::stopChild,
        &Monitor::sendChild,
    };

    The advantage of this is that it's simple, not requiring any run-time effort. The disadvantage is of course relatively complex maintenance. If for some reason Commands is modified,
24.2. THE ‘FORK’ SYSTEM CALL

s_handler must be modified as well. In cases like these, compile-time initialization often is asking for trouble. There is a simple alternative though.

- Looking at Monitor's interface we see a static data member s_initialize and a static member function initialize. The static member function handles the initialization of the s_handler array. It explicitly assigns the array's elements and any modification in ordering of enum Commands' values is automatically accounted for by recompiling initialize:

```cpp
void (Monitor::*Monitor::s_handler[int])(int, string const &);
int Monitor::initialize()
{
    s_handler[UNKNOWN] = &Monitor::unknown;
    s_handler[START] = &Monitor::createNewChild;
    s_handler(EXIT] = &Monitor::exiting;
    s_handler[STOP] = &Monitor::stopChild;
    s_handler[TEXT] = &Monitor::sendChild;
    return 0;
}
```

The member initialize is a static member and so it can be called to initialize s_initialize, a static int variable. The initialization is enforced by placing the initialization statement in the source file of a function that is known to be executed. It could be main, but if we're Monitor's maintainers and only have control over the library containing Monitor's code then that's not an option. In those cases the source file containing the destructor is a very good candidate. If a class has only one constructor and it's not defined inline then the constructor's source file is a good candidate as well. In Monitor's current implementation the initialization statement is put in run's source file, reasoning that s_handler is only needed when run is used.

24.2.7.5 The class ‘Monitor’: the member ‘run’

Monitor's core activities are performed by run. It performs the following tasks:

- Initially, the Monitor object only monitors its standard input. The set of input file descriptors to which d_selector listens is initialized to STDIN_FILENO.

- Then, in a loop d_selector's wait function is called. If input on cin is available, it is processed by processInput. Otherwise, the input has arrived from a child process. Information sent by children is processed by processChild.

- To prevent zombies, the child processes must catch their children's termination signals. This is discussed below.

  As noted by Ben Simons (ben at mrxfx dot com) Monitor must not catch the termination signals. Instead, the process spawning child processes has that responsibility (the underlying principle being that a parent process is responsible for its child processes; a child process, in turn, is responsible for its own child processes).

- As stated, run's source file also defines and initializes s_initialize to ensure the proper initialization of the s_handler array.

Here is run's implementation and s_initialize's definition:

```cpp
#include "monitor.ih"
```
int Monitor::s_initialize = Monitor::initialize();

void Monitor::run()
{
    d_selector.addReadFd(STDIN_FILENO);

    while (true)
    {
        cout << "? " << flush;
        try
        {
            d_selector.wait();

            int fd;
            while ((fd = d_selector.readFd()) != -1)
            {
                if (fd == STDIN_FILENO)
                    processInput();
                else
                    processChild(fd);
            }
        }

        cout << "NEXT ...
"
    }

    catch (char const *msg)
    {
        exiting(1, msg);
    }
}

The member function processInput reads the commands entered by the user using the program's standard input stream. The member itself is rather simple. It calls next to obtain the next command entered by the user, and then calls the corresponding function using the matching element of the s_handler[] array. Here are the members processInput and next:

    void Monitor::processInput()
    {
        string line;
        int value;
        Commands cmd = next(&value, &line);
        (this->*s_handler[cmd])(value, line);
    }

    Monitor::Commands Monitor::next(int *value, string *line)
    {
        if (!getline(cin, *line))
            exiting(1, "Monitor::next(): reading cin failed");

        if (*line == "start")
            return START;

        if (*line == "exit" || *line == "quit")
        {

    }
24.2. THE ‘FORK’ SYSTEM CALL

```cpp
*value = 0;
return EXIT;
}

if (line->find("stop") == 0)
{
    istringstream istr(line->substr(4));
    istr >> *value;
    return !istr ? UNKNOWN : STOP;
}

istringstream istr(line->c_str());
istr >> *value;
if (istr)
{
    getline(istr, *line);
    return TEXT;
}
return UNKNOWN;
}
```

All other input sensed by `d_select` is created by child processes. Because `d_select`'s `readFd` member returns the corresponding input file descriptor, this descriptor can be passed to `processChild`. Using a `IFdStreambuf` (see section 24.1.2.1), its information is read from an input stream. The communication protocol used here is rather basic. For every line of input sent to a child, the child replies by sending back exactly one line of text. This line is then read by `processChild`:

```cpp
void Monitor::processChild(int fd)
{
    IFdStreambuf ifdbuf(fd);
    istream istr(&ifdbuf);
    string line;

    getline(istr, line);
    cout << d_child[fd]->pid() << ": " << line << '\n';
}
```

The construction `d_child[fd]->pid()` used in the above source deserves some special attention. `Monitor` defines the data member `map<int, shared_ptr<Child>> d_child`. This map contains the child's order number as its key, and a (shared) pointer to the `Child` object as its value. A shared pointer is used here, rather than a `Child` object, since we want to use the facilities offered by the map, but don’t want to copy a `Child` object time and again.

24.2.7.6 The class ‘Monitor’: example

Now that `run`'s implementation has been covered, we’ll concentrate on the various commands users might enter:

- When the `start` command is issued, a new child process is started. A new element is added to `d_child` by the member `createNewChild`. Next, the `Child` object should start its activities, but the `Monitor` object cannot wait for the child process to complete its activities, as there is
no well-defined endpoint in the near future, and the user probably wants to be able to enter
more commands. Therefore, the Child process must run as a daemon. So the forked process
terminates immediately, but its own child process continues to run (in the background). Con-
sequently, createNewChild calls the child’s fork member. Although it is the child’s fork
function that is called, it is still the monitor program wherein that fork function is called. So,
the monitor program is duplicated by fork. Execution then continues:

– At the Child’s parentProcess in its parent process;
– At the Child’s childProcess in its child process

As the Child’s parentProcess is an empty function, returning immediately, the Child’s par-
ent process effectively continues immediately below createNewChild’s cp->fork() state-
ment. As the child process never returns (see section 24.2.7.7), the code below cp->fork() is
never executed by the Child’s child process. This is exactly as it should be.

In the parent process, createNewChild’s remaining code simply adds the file descriptor that’s
available for reading information from the child to the set of input file descriptors monitored
by d_selector, and uses d_child to establish the association between that file descriptor and
the Child object’s address:

```cpp
void Monitor::createNewChild(int, string const &)
{
    Child *cp = new Child(++d_nr);
    cp->fork();
    int fd = cp->readFd();
    d_selector.addReadFd(fd);
    d_child[fd].reset(cp);
    cerr << "Child " << d_nr << " started\n";
}
```

• Direct communication with the child is required for the stop <nr> and <nr> text com-
mands. The former command terminates child process <nr>, by calling stopChild. This
function locates the child process having the order number using an anonymous object of the
class Find, nested inside Monitor. The class Find simply compares the provided nr with the
children’s order number returned by their nr members:

```cpp
inline Monitor::Find::Find(int nr)
{
    d_nr(nr)
};
inline bool Monitor::Find::operator()(MapIntChild::value_type &vt) const
{
    return d_nr == vt.second->nr();
}
```

If the child process having order number nr was found, its file descriptor is removed from
d_selector’s set of input file descriptors. Then the child process itself is terminated by the
static member killChild. The member killChild is declared as a static member function,
as it is used as function argument of the for_each generic algorithm by exiting (see below).
Here is killChild’s implementation:

```cpp
void Monitor::killChild(MapIntChild::value_type it)
```
24.2. THE 'FORK' SYSTEM CALL

{
    if (kill(it.second->pid(), SIGTERM))
        cerr << "Couldn’t kill process " << it.second->pid() << ‘\n’;

    // reap defunct child process
    int status = 0;
    while( waitpid( it.second->pid(), &status, WNOHANG) > -1)
        ;
}

Having terminated the specified child process, the corresponding Child object is destroyed and its pointer is removed from d_child:

    void Monitor::stopChild(int nr, string const &)
    {
        auto it = find_if(d_child.begin(), d_child.end(), Find(nr));

        if (it == d_child.end())
            cerr << "No child number " << nr << ‘\n’;
        else
        {
            d_selector.rmReadFd(it->second->readFd());
            d_child.erase(it);
        }
    }

• The command <nr> text sends text to child process nr using the member function sendChild. This function also uses a Find object to locate the child-process having order number nr, and simply inserts the text into the writing end of a pipe connected to that child process:

    void Monitor::sendChild(int nr, string const &line)
    {
        auto it = find_if(d_child.begin(), d_child.end(), Find(nr));

        if (it == d_child.end())
            cerr << "No child number " << nr << ‘\n’;
        else
        {
            OFdnStreambuf ofdn(it->second->writeFd());
            ostream out(&ofdn);

            out << line << ‘\n’;
        }
    }

• When users enter exit or quit the member exiting is called. It terminates all child processes using the for_each generic algorithm (see section 19.1.17) to visit all elements of d_child. Then the program itself ends:

    void Monitor::exiting(int value, string const &msg)
    {
        for_each(d_child.begin(), d_child.end(), killChild);
        if (msg.length())
            cerr << msg << ‘\n’;
The program’s main function is simple and needs no further comment:

```c++
int main()
try
{
    Monitor().run();
}
catch (int exitValue)
{
    return exitValue;
}
```

24.2.7.7 The class ‘Child’

When the Monitor object starts a child process, it creates an object of the class Child. The Child class is derived from the class Fork, allowing it to operate as a daemon (as discussed in the previous section). Since Child is a daemon class, we know that its parent process must be defined as an empty function. Its childProcess member has a non-empty implementation. Here are the characteristics of the class Child:

- The Child class has two Pipe data members, to handle communications between its own child- and parent processes. As these pipes are used by the Child's child process, their names refer to the child process. The child process reads from d_in, and writes to d_out. Here is the interface of the class Child:

```c++
class Child: public Fork
{
    Pipe d_in;
    Pipe d_out;
    int d_parentReadFd;
    int d_parentWriteFd;
    int d_nr;

public:
    Child(int nr);
    virtual ~Child();
    int readFd() const;
    int writeFd() const;
    int pid() const;
    int nr() const;

private:
    virtual void childRedirections();
    virtual void parentRedirections();
    virtual void childProcess();
    virtual void parentProcess();
};
```

- The Child’s constructor simply stores its argument, a child-process order number, in its own d_nr data member:
24.2. THE ‘FORK’ SYSTEM CALL

inline Child::Child(int nr)
:
    d_nr(nr)
{}

• The Child’s child process obtains commands from its standard input stream and writes its output to its standard output stream. Since the actual communication channels are pipes, redirections must be used. The childRedirections member looks like this:

void Child::childRedirections()
{
    d_in.readFrom(STDIN_FILENO);
    d_out.writtenBy(STDOUT_FILENO);
}

• Although the parent process performs no actions, it must configure some redirections. Realizing that the names of the pipes indicate their functions in the child process. So the parent writes to d_in and reads from d_out. Here is parentRedirections:

void Child::parentRedirections()
{
    d_parentReadFd = d_out.readOnly();
    d_parentWriteFd = d_in.writeOnly();
}

• The Child object exists until it is destroyed by the Monitor’s stopChild member. By allowing its creator, the Monitor object, to access the parent-side ends of the pipes, the Monitor object can communicate with the Child’s child process via those pipe-ends. The members readFd and writeFd allow the Monitor object to access these pipe-ends:

inline int Child::readFd() const
{
    return d_parentReadFd;
}
inline int Child::writeFd() const
{
    return d_parentWriteFd;
}

• The Child object’s child process performs two tasks:
  – It must reply to information appearing at its standard input stream;
  – If no information has appeared within a certain time frame (the implementation uses an interval of five seconds), then a message is written to its standard output stream.

To implement this behavior, childProcess defines a local Selector object, adding STDIN_FILENO to its set of monitored input file descriptors.

Then, in an endless loop, childProcess waits for selector.wait() to return. When the alarm goes off it sends a message to its standard output (hence, into the writing pipe). Otherwise, it echoes the messages appearing at its standard input to its standard output. Here is the childProcess member:

void Child::childProcess()
{
    Selector selector;
size_t message = 0;
selector.addReadFd(STDIN_FILENO);
selector.setAlarm(5);

while (true)
{
    try
    {
        if (!selector.wait()) // timeout
            cout << "Child " << d_nr << " : standing by\n";
        else
        {
            string line;
            getline(cin, line);
            cout << "Child " << d_nr << ": " << ++message << ": " << line << '\n';
        }
    } catch (...) 
    {
        cout << "Child " << d_nr << ": " << ++message << ": " << 
            "select() failed" << '\n';
    }
}
exit(0);

• Two accessors are defined allowing the Monitor object to obtain the Child's process ID and its order number:

    inline int Child::pid() const
    {
        return Fork::pid();
    }
    inline int Child::nr() const
    {
        return d_nr;
    }

• A Child process terminates when the user enters a stop command. When an existing child process number was entered, the corresponding Child object is removed from Monitor's d_child map. As a result, its destructor is called. Child's destructor calls kill to terminate its child, and then waits for the child to terminate. Once its child has terminated, the destructor has completed its work and returns, thus completing the erasure from d_child. The current implementation fails if the child process doesn't react to the SIGTERM signal. In this demonstration program this does not happen. In 'real life' more elaborate killing-procedures may be required (e.g., using SIGKILL in addition to SIGTERM). As discussed in section 10.12 it is important to ensure the proper destruction. Here is the Child's destructor:

    Child::~Child()
    {
        if (pid())
        {
            cout << "Killing process " << pid() << '\n';
        }
24.3 Function objects performing bitwise operations

In section 18.1 several predefined function objects were introduced. Predefined function objects performing arithmetic operations, relational operations, and logical operations exist, corresponding to a multitude of binary- and unary operators.

Some operators appear to be missing: there appear to be no predefined function objects corresponding to bitwise operations. However, their construction is, given the available predefined function objects, not difficult. The following examples show a class template implementing a function object calling the bitwise and (\(\&\)) and a template class implementing a function object calling the unary not (\(\neg\)). It is left to the reader to construct similar function objects for other operators.

Here is the implementation of a function object calling the bitwise operators:

```cpp
#include <functional>

template <typename _Tp>
struct bitAnd: public std::binary_function<_Tp, _Tp, _Tp>
{
  _Tp operator()(_Tp const &__x, _Tp const &__y) const
  {
    return __x & __y;
  }
};
```

Here is the implementation of a function object calling \(\neg\):

```cpp
#include <functional>

template <typename _Tp>
struct bit_not: public std::unary_function<_Tp, _Tp>
{
  _Tp operator()(Tp const &__x) const
  {
    return ~__x;
  }
};
```

These and other missing predefined function objects are also implemented in the file bitfunctional, which is found in the cplusplus.yo.zip archive. These classes are derived from existing class templates (e.g., std::binary_function and std::unary_function). These base classes define several types which are expected (used) by various generic algorithms as defined in the STL (cf. chapter 19), thus following the advice offered in, e.g., the C++ header file bits/stl_function.h:
The standard functors are derived from structs named unary_function and binary_function. These two classes contain nothing but typedefs, to aid in generic (template) programming. If you write your own functors, you might consider doing the same.

Here is an example using bit_and, removing all odd numbers from a vector of int values:

```c++
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
#include "bitand.h"

using namespace std;

int main()
{
    vector<int> vi;
    for (int idx = 0; idx < 10; ++idx)
        vi.push_back(idx);

    // copy
    copy(
        vi.begin(),
        remove_if(vi.begin(), vi.end(), bind2nd(bitAnd<int>(), 1)),
        ostream_iterator<int>(cout, " ")
    );
    cout << '
';
}

/*
Generated output:
0 2 4 6 8
*/

24.4 Adding binary operators to classes

As we’ve seen in section 11.6 binary operators expecting const & arguments can be implemented in move-aware classes using a move-aware binary operator, using a rvalue reference for its first argument. This latter function can in turn be implemented using the binary assignment member. The following examples illustrated this approach for a fictitious class Binary:

```c++
class Binary
{
    public:
        Binary();
        Binary(int value);
        // copy and move constructors are available by default, or
        // they can be explicitly declared and implemented.
        Binary &operator+=(Binary const &other);    // see the text
```
24.4. ADDING BINARY OPERATORS TO CLASSES

};

inline Binary operator+(Binary &&lhs, Binary const &rhs)
{
    return std::move(lhs += rhs);  // avoids copy construction
}

inline Binary operator+(Binary const &lhs, Binary const &rhs)
{
    Binary tmp(lhs);
    return std::move(tmp) + rhs;
}

Therefore, the implementations of the binary operators eventually depend on the availability of the binary assignment operator.

Since template functions are not instantiated before their actually used we can mention a non-existing function in a template that is never instantiated. If such a function would actually be called then the compiler would generated an error message, complaining about the missing function.

This allows us to implement all binary operators, movable and non-movable, as templates. It is then only possible to call the binary operators for which a matching binary assignment exists. The template functions implementing the above addition binary operators look like this:

#ifndef INCLUDED_BINOPS_H_
#define INCLUDED_BINOPS_H_

#include <utility>

template <typename Type>
Type operator+(Type &&lhs, Type const &rhs)
{
    return lhs += rhs;
}

template <typename Type>
Type operator+(Type const &lhs, Type const &rhs)
{
    Type tmp(lhs);
    return operator+(std::move(tmp), rhs);
}

#endif

Caveat: when defining these function templates ensure that the binary operator specifying an rvalue reference as its first parameter is defined before the binary operator specifying a const lrvalue reference as its first parameter, or programs using these templates fail due to infinite recursion.

The function templates for the other binary operators can easily be added to these addition operators. After collecting them in a file binops.h include this file in, e.g., your class header file to add the binary operators to your class.

Interestingly, classes not implementing move constructors can still use these templates, as the move constructor itself is never called by the implementations of the binary operator (and its call is usually optimized away by copy elision). The following program (using modified function templates
containing output statements) behaves identically whether or not the move constructor is defined:

```cpp
#include <iostream>
using namespace std;

template <typename Class>
Class operator+(Class &&lhs, Class const &rhs)
{
    cout << "operator+(Class &&lhs, Class const &rhs)\n";
    return lhs += rhs;
}

template <typename Class>
Class operator+(Class const &lhs, Class const &rhs)
{
    cout << "operator+(Class const &, Class const &)\n";
    Class tmp(lhs);
    return operator+(std::move(tmp), rhs);
}

template <typename Class>
Class operator-(Class &&lhs, Class const &rhs)
{
    return lhs -= rhs;
}

template <typename Class>
Class operator-(Class const &lhs, Class const &rhs)
{
    Class tmp(lhs);
    return operator-(std::move(tmp), rhs);
}

class Class
{
    public:
        Class() = default;
        Class(Class const &other) = default;
        Class(int)
        {}
        Class(Class &&tmp)
        {
            cout << "Move constructor\n";
        }
        Class &operator+=(Class const &rhs)
        {
            cout << "operator+=\n";
        }
    }

    Class factory()
    {
    }
```
24.4. ADDING BINARY OPERATORS TO CLASSES

```cpp
return Class();
}

int main()
{
Class lhs;
Class rhs;
Class result;

result = lhs + rhs;
result = factory() + rhs;

// result = lhs - rhs; // this won’t compile as operator-= hasn’t been defined
}
```

24.4.1 Binary operators allowing promotions

The function templates introduced in the previous section do not support promotions. E.g., a statement like

```cpp
result = rhs + 2;
```

generates a compilation error as promotions are not recognized by the template argument deduction algorithm. Currently, the above statement needs to be rewritten to have it accepted by the compiler:

```cpp
result = rhs + Class(2);
```

If promotions are welcome, how can we change the arithmetic operator function templates so that promotions are performed? With promotions the arguments of the operator functions may be of any type, at least one of them must be of the class type offering the matching compound assignment operator. But when designing the function template we can’t say which of the two operands has that class type. So we have to specify two template types parameters for the two parameters of the operator functions. The function template must therefore start with something like this:

```cpp
template <typename LHS, typename RHS>
ReturnType operator+(LHS const &lhs, RHS const &rhs)
```

At this point we can’t yet specify `ReturnType`. It is `LHS` if `RHS` can be promoted to or is equal to `LHS`, it is `RHS` if `LHS` is promoted to `RHS`.

To determine whether `RHS` can be promoted to `LHS` we now design a simple template meta-programming class `LpromotesR` using two template type parameters, defining the value `true` (1) if the second (right-hand) type can be promoted to the first (left-hand) type, and defining the value `false` (0) if not. Here we use the same principle seen before in section 23.8.3, type convertibility:

```cpp
template <typename L, typename R>
class LpromotesR
{
struct Char2

```
In class LpromotesR the function test(L const &) is selected if R can be promoted to L, and test(...) is selected if not. The different sizes of the return types of these two test functions allows the compiler to assign 1 or 0 to the class’s enum value yes. The value yes (correctly) is 0 for R types mentioned in constructors declared with the explicit keyword, and it (correctly) is 1 if L and R happen to be the same types.

Now that we can determine whether a type can be promoted to another type, it is possible to select either LHS or RHS as the function template’s return type. If RHS can be promoted to LHS use LHS as the function template’s return type, otherwise use RHS.

Of course there is a third possibility: the LHS and RHS types cannot be used by each other’s constructors. In that case, unless there is another constructor lying around somewhere handling that situation, the compiler generates an error like:

no match for ‘operator+’ in ‘...’

Back to promotable types. We are now able to determine which type can be promoted, using LpromotesR. This yields a value that can be used as selector in the IfElse template meta-programming class template, introduced earlier (cf. section 23.2.2.2).

Now that we can select either LHS or RHS as the operator template function’s return type, we’re able to construct our arithmetic operator function template supporting promotions:

```cpp
template <typename LHS, typename RHS>
type FBB::IfElse<FBB::LpromotesR<LHS, RHS>::yes, LHS, RHS>::type
operator<<(LHS const &lhs, RHS const &rhs) {
    typedef typename FBB::IfElse<
        FBB::LpromotesR< LHS, RHS >::yes, LHS, RHS
    >::type Type;
    Type tmp(lhs);
    return std::move(tmp) << type(rhs);
}
```

The function’s return type is IfElse’s type, selected as either LHS (if RHS can be promoted to LHS) or RHS. The same trick is used in the function’s body to determine tmp’s type.

Now promotions are possible. The function template defining an rvalue reference parameter remains as-is. Together they allow the compiler to make the following decisions (using Class as the intended class name, Type as a type that is promotable to Class, and @ as the generic indication of the used operator). Unless otherwise specified the function template defining the parameter list (LHS const &lhs, RHS const &rhs) is used:

```cpp
    Class obj;
```
Type value;

obj @ obj // no promotions
obj @ Class() // same

obj @ value; // value is promoted to Class
Class() @ value; // same

value @ obj; // same
value @ Class(); // same

Class() @ obj; // calls operator@ (Class &&, Class const &)
Class() @ Class(); // same

24.5 Range-based for-loops and pointer-ranges

The standard range-based for-loop requires for its range specification an array, an initializer list, or an iterator range as offered by, e.g., containers (through their begin and end members).

Ranges defined by a pointer pair or by a subrange defined by iterator expressions cannot currently be used in combination with range-based for-loops.

The Ranger class template developed in this section defines ranges that can be used with range-based for-loops. Ranger extends the applicability of range-based for-loops by turning pointer pairs, an initial pointer or iterator and a pointer count, or a pair of iterators into a range that can be used by range-based for-loops. The Ranger class template can also be used to process a pair of reverse iterators, normally not supported by range-based for-loops.

The Ranger class template requires but one template type parameter: Iterator, representing an iterator or pointer type reaching the data when dereferenced. In practical applications users don’t have to specify Ranger’s template type. The function template ranger deduces the required Iterator type and returns the appropriate Ranger object.

The ranger function template can be used in various ways:

- Ranger<Iterator> ranger(Iterator const &begin, Iterator const &end) this function template returns a Ranger object for the (sub)range defined by two (reverse) iterators. Its definition is:

  template <typename Iter>
  Ranger<Iter> ranger(Iter &&begin, Iter &&end)
  {
    return Ranger<Iter>(begin, end);
  }

- Ranger<Iterator> ranger(Iterator const &begin, size_t count) this function template returns a Ranger object for the (sub)range defined by the (reverse) iterator range begin and begin + count. Its definition is:

  template <typename Data>
  Ranger<Data *> ranger(Data *begin, Data *end)
  {
    return Ranger<Data *> (begin, end);
  }
• Ranger<Data> ranger(Data *begin, Data *end) this function template returns a Ranger object for the (sub)ranging defined by the two pointers begin and end. Its definition is:

    template <typename Iter>
    Ranger<Iter> ranger(Iter &&begin, size_t count)
    {
        return Ranger<Iter>(begin, begin + count);
    }

• Ranger<Data> ranger(Data *begin, size_t count) this function template returns a Ranger object for the (sub)ranging defined by the two pointers begin and begin + count. Its definition is:

    template <typename Data>
    Ranger<Data *> ranger(Data *begin, size_t count)
    {
        return Ranger<Data *>(begin, begin + count);
    }

The Ranger class template itself offers a constructor expecting two Iterator const & parameters, where Iterator is Ranger's template type parameter. Although named 'Iterator' it can also be a pointer to some data type (e.g., std::string *).

The class only needs two members, begin and end, since these are the only members called by range-based for-loops. These members can be const members, returning Iterator const references. This also is the required return type if Iterator itself was a pointer type (like int *). Since a 'Iterator const &' does not imply that the dereferenced Iterator is immutable, the data to which the iterator returned by begin() can actually be modified, if Iterator unless Iterator is a Type const * or a const_iterator type.

If reverse iterators are passed to Ranger's constructor (the reversed begin iterator should be passed as Ranger constructor's first argument, the reversed end iterator as its second argument), then Ranger's begin and end members return reverse iterators. Since the intended use of Ranger objects is to define a range for range-base for-loops, members like rbegin and rend were omitted from Ranger's interface.

Here is Ranger's implementation (using in-class implementations for brevity):

    template <typename Iter>
    class Ranger
    {
        Iter d_begin;
        Iter d_end;

        public:
        Ranger(Iter const &begin, Iter const &end)
            :
                d_begin(begin),
                d_end(end)
        {}

        Iter const &begin() const
        {
            return d_begin;
        }
Using `ranger` is easy. Here is an example of a program displaying a program’s command-line arguments using a range-based for-loop:

```cpp
int main(int argc, char**argv)
{
    for (auto ptr: ranger(argv, argc))
        cout << ptr << ‘\n’;
}
```

### 24.6 Distinguishing lvalues from rvalues with `operator[]( )`

A problem with `operator[]` is that it can’t distinguish between its use as an lvalue and as an rvalue. It is a familiar misconception to think that

Type const &operator[](size_t index) const

is used as rvalue (as the object isn’t modified), and that

Type &operator[](size_t index)

is used as lvalue (as the returned value can be modified).

The compiler, however, distinguishes between the two operators only by the const-status of the object for which `operator[]` is called. With const objects the former operator is called, with non-const objects the latter is always used. It is always used, irrespective of it being used as lvalue or rvalue.

Being able to distinguish between lvalues and rvalues can be very useful. Consider the situation where a class supporting `operator[]` stores data of a type that is very hard to copy. With data like that reference counting (e.g., using `shared_ptr`s) is probably used to prevent needless copying.

As long as `operator[]` is used as rvalue there’s no need to copy the data, but the information must be copied if it is used as lvalue.

The Proxy Design Pattern (cf. Gamma et al. (1995)) can be used to distinguish between lvalues and rvalues. With the Proxy Design Pattern an object of another class (the Proxy class) is used to act as a stand in for the ‘real thing’. The proxy class offers functionality that cannot be offered by the data themselves, like distinguishing between its use as lvalue or rvalue. A proxy class can be used in many situations where access to the real data cannot or should not be directly provided. In this regard iterator types are examples of proxy classes as they create a layer between the real data and the software using the data. Proxy classes could also dereference pointers in a class storing its data by pointers.
In this section we concentrate on the distinction between using operator[] as lvalue and rvalue. Let’s assume we have a class Lines storing lines from a file. Its constructor expects the name of a stream from which the lines are read and it offers a non-const operator[](that can be used as lvalue or rvalue (the const version of operator[] is omitted as it causes no confusion because it is always used as rvalue):

```cpp
class Lines
{
    std::vector<std::string> d_line;

public:
    Lines(std::istream &in);
    std::string &operator[](size_t idx);
};
```

To distinguish between lvalues and rvalues we must find distinguishing characteristics of lvalues and rvalues that we can exploit. Such distinguishing characteristics are operator=(which is always used as lvalue) and the conversion operator (which is always used as rvalue). Rather than having operator[] return a string & we can let it return a Proxy object that is able to distinguish between its use as lvalue and rvalue.

The class Proxy thus needs operator=(string const &other) (acting as lvalue) and operator std::string const &() const (acting as rvalue). Do we need more operators? The std::string class also offers operator+=, so we should probably implement that operator as well. Plain characters can also be assigned to string objects (even using their numeric values). As string objects cannot be constructed from plain characters promotion cannot be used with operator=(string const &other) if the right-hand side argument is a character. Implementing operator=(char value) could therefore also be considered. These additional operators are left out of the current implementation but ‘real life’ proxy classes should consider implementing these additional operators as well. Another subtlety is that Proxy’s operator std::string const &() const is not used when using ostream’s insertion operator or istream’s extraction operator as these operators are implemented as templates not recognizing our Proxy class type. So when stream insertion and extraction is required (it probably is) then Proxy must be given its own overloaded insertion and extraction operator. Here is an implementation of the overloaded insertion operator inserting the object for which Proxy is a stand-in:

```cpp
inline std::ostream &operator<<(std::ostream &out, Lines::Proxy const &proxy)
{
    return out << static_cast<std::string const &>(proxy);
}
```

There’s no need for any code (except Lines) to create or copy Proxy objects. Proxy’s constructor should therefore be made private, and Proxy can declare Lines to be its friend. In fact, Proxy is intimately related to Lines and can be defined as a nested class. In the revised Lines class operator[] no longer returns a string but instead a Proxy is returned. Here is the revised Lines class, including its nested Proxy class:

```cpp
class Lines
{
    std::vector<std::string> d_line;

public:
    class Proxy;
```
Proxy operator[](size_t idx);
class Proxy
{
    friend Proxy Lines::operator[](size_t idx);
    std::string &d_str;
    Proxy(std::string &str);
public:
    std::string &operator=(std::string const &rhs);
    operator std::string const &() const;
};
Lines(std::istream &in);

Proxy's members are very lightweight and can usually be implemented inline:

inline Lines::Proxy::Proxy(std::string &str)
{
    d_str(str)
}
inline std::string &Lines::Proxy::operator=(std::string const &rhs)
{
    return d_str = rhs;
}
inline Lines::Proxy::operator std::string const &() const
{
    return d_str;
}

The member Lines::operator[] can also be implemented inline: it merely returns a Proxy object initialized with the string associated with index idx.

Now that the class Proxy has been developed it can be used in a program. Here is an example using the Proxy object as lvalue or rvalue. On the surface Lines objects won't behave differently from Lines objects using the original implementation, but by adding an identifying cout statement to Proxy's members it can be shown that operator[] behaves differently when used as lvalue or as rvalue:

int main()
{
    ifstream in("lines.cc");
    Lines lines(in);

    string s = lines[0]; // rvalue use
    lines[0] = s; // lvalue use
    cout << lines[0] << \n'; // rvalue use
    lines[0] = "hello world";} // lvalue use
    cout << lines[0] << \n'; // rvalue use
24.7 Implementing a ‘reverse_iterator’

In section 22.14.1 the construction of iterators and reverse iteraters was discussed. In that section the iterator was constructed as an inner class in a class derived from a vector of pointers to strings.

An object of this nested iterator class handles the dereferencing of the pointers stored in the vector. This allowed us to sort the strings pointed to by the vector’s elements rather than the pointers.

A drawback of this is that the class implementing the iterator is closely tied to the derived class as the iterator class was implemented as a nested class. What if we would like to provide any class derived from a container class storing pointers with an iterator handling pointer-dereferencing?

In this section a variant of the earlier (nested class) approach is discussed. Here the iterator class is defined as a class template, not only parameterizing the data type to which the container’s elements point but also the container’s iterator type itself. Once again, we concentrate on developing a RandomIterator as it is the most complex iterator type.

Our class is named RandomPtrIterator, indicating that it is a random iterator operating on pointer values. The class template defines three template type parameters:

- The first parameter specifies the derived class type (Class). Like before, RandomPtrIterator’s constructor is private. Therefore friend declarations are needed to allow client classes to construct RandomPtrIterators. However, a friend class Class cannot be used as template parameter types cannot be used in friend class ... declarations. But this is a minor problem as not every member of the client class needs to construct iterators. In fact, only Class’s begin and end members must construct iterators. Using the template’s first parameter, friend declarations can be specified for the client’s begin and end members.

- The second template parameter parameterizes the container’s iterator type (BaseIterator);

- The third template parameter indicates the data type to which the pointers point (Type).

RandomPtrIterator has one private data member, a BaseIterator. Here is the class interface and the constructor’s implementation:

```cpp
#include <iterator>

template <typename Class, typename BaseIterator, typename Type>
class RandomPtrIterator:
  public std::iterator<std::random_access_iterator_tag, Type>
{
  friend RandomPtrIterator<Class, BaseIterator, Type> Class::begin();
  friend RandomPtrIterator<Class, BaseIterator, Type> Class::end();

  BaseIterator d_current;

  RandomPtrIterator(BaseIterator const &current);

  public:
  bool operator!=(RandomPtrIterator const &other) const;
  int operator-(RandomPtrIterator const &rhs) const;
  RandomPtrIterator operator+(int step) const;
  Type &operator*() const;
  bool operator<(RandomPtrIterator const &other) const;
}
```
24.7. IMPLEMENTING A ‘REVERSE_ITERATOR’

RandomPtrIterator &operator--();
RandomPtrIterator operator--(int);
RandomPtrIterator &operator++();
RandomPtrIterator operator++(int);
bool operator==(RandomPtrIterator const &other) const;
RandomPtrIterator operator-(int step) const;
RandomPtrIterator &operator-=(int step);
RandomPtrIterator &operator+=(int step);
Type *operator->() const;

Looking at its friend declarations, we see that the members begin and end of a class Class, returning a RandomPtrIterator object for the types Class, BaseIterator and Type are granted access to RandomPtrIterator’s private constructor. That is exactly what we want. The Class’s begin and end members are declared as bound friends.

All RandomPtrIterator’s remaining members are public. Since RandomPtrIterator is just a generalization of the nested class iterator developed in section 22.14.1, re-implementing the required member functions is easy and only requires us to change iterator into RandomPtrIterator and to change std::string into Type. For example, operator<, defined in the class iterator as

```cpp
inline bool StringPtr::iterator::operator<(iterator const &other) const
{
    return d_current < other.d_current;
}
```

is now implemented as:

```cpp
template <typename Class, typename BaseIterator, typename Type>
bool RandomPtrIterator<Class, BaseIterator, Type>::operator<( RandomPtrIterator const &other) const
{
    return **d_current < **other.d_current;
}
```

Some additional examples: operator*, defined in the class iterator as

```cpp
inline std::string &StringPtr::iterator::operator*() const
{
    return **d_current;
}
```

is now implemented as:

```cpp
template <typename Class, typename BaseIterator, typename Type>
std::string &RandomPtrIterator<Class, BaseIterator, Type>::operator*() const
{
    return **d_current;
}
```
Type &RandomPtrIterator<Class, BaseIterator, Type>::operator*() const
{
    return **d_current;
}

The pre- and postfix increment operators are now implemented as:

template <typename Class, typename BaseIterator, typename Type>
RandomPtrIterator<Class, BaseIterator, Type>& RandomPtrIterator<Class, BaseIterator, Type>::operator++()
{
    ++d_current;
    return *this;
}
template <typename Class, typename BaseIterator, typename Type>
RandomPtrIterator<Class, BaseIterator, Type> RandomPtrIterator<Class, BaseIterator, Type>::operator++(int)
{
    return RandomPtrIterator(d_current++);
}

Remaining members can be implemented accordingly, their actual implementations are left as exercises to the reader (or can be obtained from the cplusplus.yo.zip archive, of course).

Re-implementing the class StringPtr developed in section 22.14.1 is not difficult either. Apart from including the header file defining the class template RandomPtrIterator, it only requires a single modification. Its iterator typedef must now be associated with a RandomPtrIterator. Here is the full class interface and the class's inline member definitions:

```cpp
#ifndef INCLUDED_STRINGPTR_H_
#define INCLUDED_STRINGPTR_H_

#include <vector>
#include <string>
#include "iterator.h"

class StringPtr: public std::vector<std::string *>
{
    public:
        typedef RandomPtrIterator
        <
            StringPtr,
            std::vector<std::string *>::iterator,
            std::string
        > iterator;
        typedef std::reverse_iterator<iterator> reverse_iterator;

        iterator begin();
        iterator end();
        reverse_iterator rbegin();
        reverse_iterator rend();
};
```
```cpp
inline StringPtr::iterator StringPtr::begin()
{
    return iterator(this->std::vector<std::string *>::begin());
}
inline StringPtr::iterator StringPtr::end()
{
    return iterator(this->std::vector<std::string *>::end());
}
inline StringPtr::reverse_iterator StringPtr::rbegin()
{
    return reverse_iterator(end());
}
inline StringPtr::reverse_iterator StringPtr::rend()
{
    return reverse_iterator(begin());
}
#endif
```

Including StringPtr's modified header file into the program given in section 22.14.2 results in a program behaving identically to its earlier version. In this case StringPtr::begin and StringPtr::end return iterator objects constructed from a template definition.

## 24.8 Using ‘bisonc++’ and ‘flexc++’

The example discussed below digs into the peculiarities of using parser- and scanner generators generating C++ sources. Once the input for a program exceeds a certain level of complexity, it becomes attractive to use scanner- and parser-generators generating the code which does the actual input recognition.

The examples in this and subsequent sections assume that the reader knows how to use the scanner generator flex and the parser generator bison. Both bison and flex are well documented elsewhere. The original predecessors of bison and flex, called yacc and lex are described in several books, e.g. in O'Reilly's book 'lex & yacc'.

Scanner- and parser generators are also available as free software. Both bison and flex are usually part of software distributions or they can be obtained from ftp://prep.ai.mit.edu/pub/non-gnu. Flex creates a C++ class when `%option c++` is specified.

For parser generators the program bison is available. In the early 90's Alain Coetmeur (coetmeur@icdc.fr) created a C++ variant (bison++) creating a parser class. Although the bison++ program produces code that can be used in C++ programs it also shows many characteristics that are more suggestive of a C context than a C++ context. In January 2005 I rewrote parts of Alain's bison++ program, resulting in the original version of the program bisonc++. Then, in May 2005 a complete rewrite of the bisonc++ parser generator was completed (version number 0.98). Current versions of bisonc++ can be downloaded from http://bisoncpp.sourceforge.net/, where it is available as source archive and as binary (i386) Debian package (including bisonc++'s documentation).
Bisonc++ creates a cleaner parser class than bison++. In particular, it derives the parser class from a base-class, containing the parser’s token- and type-definitions as well as all member functions which should not be (re)defined by the programmer. As a result of this approach, the generated parser class is very small, declaring only members that are actually defined by the programmer (as well as some other members, generated by bisonc++ itself, implementing the parser’s parse() member). One member that is not implemented by default is lex, producing the next lexical token. When the directive %scanner (see section 24.8.2.1) is used, bisonc++ produces a standard implementation for this member; otherwise it must be implemented by the programmer.

In early 2012 the program flexc++ http://flexcpp.org/ reached its initial release. Like bisonc++ it is part of the Debian linux distribution6.

Jean-Paul van Oosten (j.p.van.oosten@rug.nl<j.p.van.oosten@rug.nl>) and Richard Berendsen (richardberendsen@xs4all.nl<richardberendsen@xs4all.nl>) started the flexc++ project in 2008 and the final program was completed by Jean-Paul and me between 2010 and 2012.

These sections of the C++ Annotations focus on bisonc++ as our parser generator and flexc++ as our lexical scanner generator. Previous releases of the C++ Annotations were using flex as the scanner generator.

Using flex++ and bison++ class-based scanners and parsers are generated. The advantage of this approach is that the interface to the scanner and the parser tends to become cleaner than without using class interfaces. Furthermore, classes allow us to get rid of most if not all global variables, making it easy to use multiple parsers in one program.

Below two example programs are developed. The first example only uses flexc++. The generated scanner monitors the production of a file from several parts. That example focuses on the lexical scanner and on switching files while churning through the information. The second example uses both flexc++ and bisonc++ to generate a scanner and a parser transforming standard arithmetic expressions to their postfix notations, commonly used in code generated by compilers and in HP-calculators. In the second example the emphasis is mainly on bisonc++ and on composing a scanner object inside a generated parser.

### 24.8.1 Using ‘flexc++’ to create a scanner

The lexical scanner developed in this section is used to monitor the production of a file from several subfiles. The setup is as follows: the input-language defines #include directives, followed by a text string specifying the file (path) which should be included at the location of the #include.

In order to avoid complexities irrelevant to the current example, the format of the #include statement is restricted to the form #include <filepath>. The file specified between the pointed brackets should be available at the location indicated by filepath. If the file is not available, the program terminates after issuing an error message.

The program is started with one or two filename arguments. If the program is started with just one filename argument, the output is written to the standard output stream cout. Otherwise, the output is written to the stream whose name is given as the program’s second argument.

The program defines a maximum nesting depth. Once this maximum is exceeded, the program terminates after issuing an error message. In that case, the filename stack indicating where which file was included is printed.

An additional feature of the program is that (standard C++) comment-lines are ignored. Include-directives in comment-lines are also ignored.

6http://www.debian.org
24.8. USING 'BISONC++' AND 'FLEXC++'

The program is created in five major steps:

- First, the file lexer is constructed, containing the input-language specifications.
- From the specifications in lexer the requirements for the class Scanner evolve. The Scanner class derives from the base class ScannerBase generated by flexc++.
- Next, main is constructed. A Scanner object is created inspecting the command-line arguments. If successful, the scanner's member lex is called to produce the program's output.
- Now that the global setup of the program has been specified, the member functions of the various classes are implemented.
- Finally, the program is compiled and linked.

24.8.1.1 The derived class ‘Scanner’

The function matching the regular expression rules (lex) is a member of the class Scanner. Since Scanner is derived from ScannerBase, it has access to all of ScannerBase’s protected members that execute the lexical scanner’s regular expression matching algorithm.

Looking at the regular expressions themselves, notice that we need rules to recognize comment, #include directives, and all remaining characters. This all is fairly standard practice. When an #include directive is sensed, the directive is parsed by the scanner. This too is common practice. Our lexical scanner performs the following tasks:

- As usual, preprocessor directives are not analyzed by a parser, but by the lexical scanner;
- The scanner uses a mini scanner to extract the filename from the directive, throwing an exception if this fails;
- If the filename could be extracted, processing switches to the next stream, controlling for a maximum nesting depth.
- Once the end of the current file has been reached processing automatically returns to the previous file, restoring the previous file name an line number. The scanner returns 0 if all files have been processed.

24.8.1.2 The lexical scanner specification file

The lexical scanner specification file is organized comparably to the one used for flex in C contexts. However, in C++ contexts, flexc++ creates a class Scanner, rather than just a scanner function.

Flexc++’s specification file consists of two sections:

- The specification file’s first section is flexc++’s symbol area, used to define symbols, like a mini scanner, or options. The following options are suggested:
  - %debug: includes debugging code into the code generated by flexc++. Calling the member function setDebug(true) activates this debugging code at run-time. When activated, information about the matching process is written to the standard output stream. The execution of debug code is suppressed after calling the member function setDebug(false).
- %filenames: defines the base-name of the class header files generated by flexc++. By default the class name (itself using the default Scanner) is used.

Here is the specification files' symbol area:

%filenames scanner
%debug
%max-depth 3

%x comment
%x include

- The specification file's second section is a rules section in which the regular expressions and their associated actions are defined. In the example developed here, the lexer should copy information from the standard input stream (std::cin) to the standard output stream (std::cout). For this the predefined macro ECHO can be used. Here are the rules:

```%
  // The comment-rules: comment is ignored.
  // ignore eoln comment
  ":" begin(StartCondition::comment);
<comment>{
    \n    // ignore all characters in std C comment
    ":" begin(StartCondition::INITIAL);
}

  // File switching: #include <filepath>
#include[ \t ]+"<" begin(StartCondition::include);
<include>{
    \n    // File switching: #include <filepath>
    ":"[ \t ]*\n    switchSource();
    \n    throw runtime_error("Invalid include statement");
}

  // The default rule: echo anything else to std::cout
  .\n  echo();
```

24.8.1.3 Implementing ‘Scanner’

The class Scanner is generated once by flexc++. This class has access to several members defined by its base class ScannerBase. Some of these members have public access rights and can be used by code external to the class Scanner. These members are extensively documented in the flexc++(1) man-page, and the reader is referred to this man-page for further information.

Our scanner performs the following tasks:

- it matches regular expressions, ignoring comment, and writing the matched text to the standard output stream;
- it switches to other files, and returns to the previous file once a file has completely been processed, ending the lexical scan once the end of the first input file has been reached.

The #include statements in the input allow the scanner to distill the name of the file where the scanning process must continue. This file name is stored in a local variable d_nextSource and a
member stackSource handles the switch to the next source. Nothing else is required. Pushing and popping input files is handled by the scanner's members pushStream and popStream, provided by flexc++. Scanner's interface, therefore, only needs one additional function declaration: switchSource.

Switching streams is handled as follows: once the scanner has extracted a filename from an #include directive, a switch to another file is realized by switchSource. This member calls pushStream, defined by flexc++, to stack the current input stream and to switch to the stream whose name is stored in d_nextSource. This also ends the include mini-scanner, so to return the scanner to its default scanning mode begin(StartCondition::__::INITIAL) is called. Here is its source:

```cpp
#include "scanner.ih"

void Scanner::switchSource()
{
    pushStream(d_nextSource);
    begin(StartCondition::__::INITIAL);
}
```

The member pushStream, defined by flexc++, handles all necessary checks, throwing an exception if the file could not be opened or if too many files are stacked.

The member performing the lexical scan is defined by flexc++ in Scanner::lex, and this member can be called by code to process the tokens returned by the scanner.

24.8.1.4 Using a ‘Scanner’ object

The program using our Scanner is very simple. It expects a filename indicating where to start the scanning process.

The program first checks the number of arguments. If at least one argument was given, then that argument is passed to Scanner's constructor, together with a second argument "-", indicating that the output should go to the standard output stream.

If the program receives more than one argument debug output, extensively documenting the lexical scanner's actions, is written to the standard output stream as well.

Next the Scanner's lex member is called. If anything fails, a std::exception is thrown, which is caught by main's try-block's catch clause. Here is the program's source:

```cpp
#include "lexer.ih"

int main(int argc, char **argv)
try
{
    if (argc == 1)
    {
        cerr << "Filename argument required\n";
        return 1;
    }

    Scanner scanner(argv[1], "-");
```
24.8.1.5 Building the program

The final program is constructed in two steps. These steps are given for a Unix system, on which 
flexc++ and the Gnu C++ compiler g++ have been installed:

- First, the lexical scanner’s source is created using flexc++. For this the following command 
can be given:

```
flexc++ lexer
```

- Next, all sources are compiled and linked:

```
g++ --std=c++11 -Wall *.cc
```

Flexc++ can be downloaded from http://flexcpp.sf.net/, and requires the bobcat library, 
which can be downloaded from http://bobcat.sf.net/.

24.8.2 Using ‘bisonc++’ and ‘flexc++’

Once an input language exceeds a certain level of complexity, a parser is often used to control the 
complexity of the language. In this case, a parser generator can be used to generate the code verify-
ning the input’s grammatical correctness. The lexical scanner (preferably composed into the parser) 
provides chunks of the input, called tokens. The parser then processes the series of tokens generated 
by the lexical scanner.

Starting point when developing programs that use both parsers and scanners is the grammar. The 
grammar defines a set of tokens that can be returned by the lexical scanner (called the scanner 
below).

Finally, auxiliary code is provided to ‘fill in the blanks’: the actions performed by the parser and 
by the scanner are not normally specified literally in the grammar rules or lexical regular expres-
sions, but should be implemented in member functions, called from the parser’s rules or which are 
associated with the scanner’s regular expressions.

In the previous section we’ve seen an example of a C++ class generated by flexc++. In the current 
section we concentrate on the parser. The parser can be generated from a grammar specification 
file, processed by the program bisonc++. The grammar specification file required by bisonc++ is 
similar to the file processed by bison (or bison++, bisonc++’s predecessor, written in the early 
nineties by Alain Coetmeur).

In this section a program is developed converting infix expressions, where binary operators are 
written between their operands, to postfix expressions, where operators are written behind their
operands. Also, the unary operator - is converted from its prefix notation to a postfix form. The unary + operator is ignored as it requires no further actions. In essence our little calculator is a micro compiler, transforming numeric expressions into assembly-like instructions.

Our calculator recognizes a rather basic set of operators: multiplication, addition, parentheses, and the unary minus. We'll distinguish real numbers from integers, to illustrate a subtlety in bison-like grammar specifications. That's all. The purpose of this section is, after all, to illustrate the construction of a C++ program that uses both a parser and a lexical scanner, rather than to construct a full-fledged calculator.

In the coming sections we'll develop the grammar specification for bison++. Then, the regular expressions for the scanner are specified. Following that, the final program is constructed.

24.8.2.1 The ‘bison++’ specification file

The grammar specification file required by bison++ is comparable to the specification file required by bison. Differences are related to the class nature of the resulting parser. Our calculator distinguishes real numbers from integers, and supports a basic set of arithmetic operators.

Bison++ should be used as follows:

- As usual, a grammar is defined. With bison++ this is no different, and bison++ grammar definitions are for all practical purposes identical to bison's grammar definitions.

- Having specified the grammar and (usually) some declarations bison++ can generate files defining the parser class and the implementation of the member function parse.

- All class members (except those that are required for the proper functioning of the member parse) must be separately implemented. Of course, they should also be declared in the parser class's header. At the very least the member lex must be implemented. This member is called by parse to obtain the next available token. However, bison++ offers a facility providing a standard implementation of the function lex. The member function error(char const *msg) is given a simple default implementation that may be modified by the programmer. The member function error is called when parse detects (syntactic) errors.

- The parser can now be used in a program. A very simple example would be:

```c
int main()
{
    Parser parser;
    return parser.parse();
}
```

The bison++ specification file has two sections:

- The declaration section. In this section bison’s tokens, and the priority rules for the operators are declared. However, bison++ also supports several new declarations. These new declarations are important and are discussed below.

- The rules section. The grammatical rules define the grammar. This section is identical to the one required by bison, albeit that some members that were available in bison and bison++ are obsolete in bison++, while other members can be used in a wider context. For example, ACCEPT and ABORT can be called from any member called from the parser's action blocks to terminate the parsing process.
Readers familiar with bison may note that there is no header section anymore. Header sections are used by bison to provide for the necessary declarations allowing the compiler to compile the C function generated by bison. In C++ declarations are part of or already used by class definitions. Therefore, a parser generator generating a C++ class and some of its member functions does not require a header section anymore.

The declaration section The declaration section contains several sets of declarations, among which definitions of all the tokens used in the grammar and the priorities and associativities of the mathematical operators. Moreover, several new and important specifications can be used here as well. Those relevant to our current example and only available in bisonc++ are discussed here. The reader is referred to bisonc++’s man-page for a full description.

- **%baseclass-preinclude** header
  Use header as the pathname to the file pre-included in the parser’s base-class header. This declaration is useful in situations where the base class header file refers to types which might not yet be known. E.g., with %union a std::string * field might be used. Since the class std::string might not yet be known to the compiler once it processes the base class header file we need a way to inform the compiler about these classes and types. The suggested procedure is to use a pre-include header file declaring the required types. By default header is surrounded by double quotes (using, e.g., #include "header"). When the argument is surrounded by angle brackets #include <header> is included. In the latter case, quotes might be required to escape interpretation by the shell (e.g., using -H ’<header>’).

- **%filenames** header
  Defines the generic name of all generated files, unless overridden by specific names. By default the generated files use the class-name as the generic file name.

- **%scanner** header
  Use header as the pathname to the file pre-included in the parser’s class header. This file should define a class Scanner, offering a member int lex() producing the next token from the input stream to be analyzed by the parser generated by bisonc++. When this option is used the parser’s member int lex() is predefined as (assuming the default parser class name Parser is used):

  ```cpp
  inline int Parser::lex()
  {
      return d_scanner.lex();
  }
  ```

  and an object Scanner d_scanner is composed into the parser. The d_scanner object is constructed by its default constructor. If another constructor is required, the parser class may be provided with an appropriate (overloaded) parser constructor after having constructed the default parser class header file using bisonc++. By default header is surrounded by double quotes (using, e.g., #include "header"). When the argument is surrounded by angle brackets #include <header> is included.

- **%stype** typename
  The type of the semantic value of tokens. The specification typename should be the name of an unstructured type (e.g., size_t). By default it is int. See YYSTYPE in bison. It should not be used if a %union specification is used. Within the parser class, this type may be used as STYPE.

- **%union** union-definition
  Acts identically to the bison declaration. As with bison this generates a union for the parser’s semantic type. The union type is named STYPE. If no %union is declared, a simple stack-type
may be defined using the %stype declaration. If no %stype declaration is used, the default stacktype (int) is used.

An example of a %union declaration is:

```c
%union
{
    int i;
    double d;
};
```

In pre-C++11 code a union cannot contain objects as its fields, as constructors cannot be called when a union is created. This means that a string cannot be a member of the union. A string *, however, is a possible union member. It might also be possible to use unrestricted unions (cf. section 12.6), having class type objects as fields.

As an aside: the scanner does not have to know about such a union. It can simply pass its scanned text to the parser through its matched member function. For example using a statement like

```c
$$\cdot i = A2x(d\_scanner\_matched());
```

matched text is converted to a value of an appropriate type.

Tokens and non-terminals can be associated with union fields. This is strongly advised, as it prevents type mismatches, since the compiler may then check for type correctness. At the same time, the bison specific variables $$, $1, $2, etc. may be used, rather than the full field specification (like $$\cdot i). A non-terminal or a token may be associated with a union field using the <fieldname> specification. E.g.,

```c
%token <i> INT // token association (deprecated, see below)
<d> DOUBLE
%type <i> intExpr // non-terminal association
```

In the example developed here, both the tokens and the non-terminals can be associated with a union field. However, as noted before, the scanner does not have to know about all this. In our opinion, it is cleaner to let the scanner do just one thing: scan texts. The parser, knowing what the input is all about, may then convert strings like "123" to an integer value. Consequently, the association of a union field and a token is discouraged. Below, while describing the grammar's rules, this is further illustrated.

In the %union discussion the %token and %type specifications should be noted. They are used to specify the tokens (terminal symbols) that can be returned by the scanner, and to specify the return types of non-terminals. Apart from %token the token declarators %left, %right, and %nonassoc can be used to specify the associativity of operators. The tokens mentioned at these indicators are interpreted as tokens indicating operators, associating in the indicated direction. The precedence of operators is defined by their order: the first specification has the lowest priority. To overrule a certain precedence in a certain context %prec can be used. As all this is standard bison++ practice, it isn’t further elaborated here. The documentation provided with bison++’s distribution should be consulted for further reference.

Here is the specification of the calculator’s declaration section:

```c
%filenames parser
```
In the declaration section, \%type specifiers are used, associating the intExpr rule's value (see the next section) to the i-field of the semantic-value union, and associating doubleExpr's value to the d-field. This approach, admittedly, is rather complex, as expression rules must be included for each of the supported union types. Alternatives are definitely possible, and are illustrated in subsequent sections (e.g., section 24.8.3).

The grammar rules The rules and actions of the grammar are specified as usual. The grammar for our little calculator is given below. There are quite a few rules, but they illustrate various features offered by bison++.

The implementation of that function is simple, but it is worth while noting that it calls Parser::ACCEPT, showing that ACCEPT can be called indirectly from a production rule's action block. Here are the grammar's production rules:
24.8. USING 'BISONC++' AND 'FLEXC++'
This grammar is used to implement a simple calculator in which integer and real values can be negated, added, and multiplied and in which standard priority rules can be overruled by parentheses. The grammar shows the use of typed nonterminal symbols: doubleExpr is linked to real (double) values, intExpr is linked to integer values. Precedence and type association is defined in the parser's definition section.

The **Parser's header file** Several class members called from the grammar are defined as member templates. Bisonc++ generates multiple files, among which the file defining the parser's class. Functions called from the production rule's action blocks are usually member functions of the parser. These member functions must be declared and defined. Once bisonc++ has generated the header file defining the parser's class, that header file isn't automatically rewritten, allowing the programmer to add new members to the parser class whenever required. Here is 'parser.h' as used in our little calculator:

```cpp
#ifndef Parser_h_included
#define Parser_h_included

#include <iostream>
#include <sstream>
#include <bobcat/a2x>
```

```cpp
doubleExpr ' + ' doubleExpr
{
    $$ = exec('+', $1, $3);
}
| doubleExpr ' + ' intExpr
{
    $$ = exec('+', $1, d($3));
}
| intExpr ' + ' doubleExpr
{
    $$ = exec('+', d($1), $3);
}
| '(' doubleExpr ')'
{
    $$ = $2;
}
| ' - ' doubleExpr %prec UnaryMinus
{
    $$ = neg($2);
}
| DOUBLE
{
    $$ = convert<double>();
}
;
```
#include "parserbase.h"
#include "../scanner/scanner.h"

#undef Parser

class Parser: public ParserBase
{
    std::ostringstream d_rpn;
    // $insert scannerobject
    Scanner d_scanner;

    public:
    int parse();

    private:
    template <typename Type>
    Type exec(char c, Type left, Type right);

    template <typename Type>
    Type neg(Type op);

    template <typename Type>
    Type convert();

    void display(int x);
    void display(double x);
    void done() const;
    void reset();
    void error(char const* msg);
    int lex();
    void print();

    static double d(int i);

    // support functions for parse():
    void executeAction(int d_ruleNr);
    void errorRecovery();
    int lookup(bool recovery);
    void nextToken();
    void print__();
};

inline double Parser::d(int i)
{
    return i;
}

template <typename Type>
Type Parser::exec(char c, Type left, Type right)
{
    d_rpn << " " << c << " ";
    return c == '*' ? left * right : left + right;
}


```cpp
template <typename Type>
Type Parser::neg(Type op)
{
    d_rpn << " n ";
    return -op;
}

template <typename Type>
Type Parser::convert()
{
    Type ret = FBB::A2x(d_scanner.matched());
    d_rpn << " " << ret << " ";
    return ret;
}

inline void Parser::error(char const *msg)
{
    std::cerr << msg << '\n';
}

inline int Parser::lex()
{
    return d_scanner.lex();
}

inline void Parser::print()
{
}

#endif

24.8.2.2 The 'flexc++' specification file

The flex-specification file used by the calculator is simple: blanks are ignored, single characters are returned, and numeric values are returned as either Parser::INT or Parser::DOUBLE tokens.

The flexc++ directive %interactive is provided since the calculator is a program actively interacting with its human user.

Here is the complete flexc++ specification file:

```
24.8. USING ‘BISONC++’ AND ‘FLEXC++’

24.8.3 Building the program

The calculator is built using bisonc++ and flexc++. Here is the implementation of the calculator’s main function:

```cpp
#include "parser/parser.h"

using namespace std;

int main()
{
    Parser parser;

    cout << "Enter (nested) expressions containing ints, doubles, *, + and "
         "unary -,\n"         "operators. Enter an empty line to stop.\n";

    return parser.parse();
}
```

The parser’s files parse.cc and parserbase.h are generated by the command:

```
bisonc++ grammar
```

The file parser.h is created only once, to allow the developer to add members to the Parser class once the need for them arises.

The program flexc++ is used to create a lexical scanner:

```
flexc++ lexer
```

On Unix systems a command like

```
g++ --std=c++11 -Wall -o calc *.cc -lbobcat -s
```

can be used to compile and link the source of the main program and the sources produced by the scanner and parser generators. The example uses the A2x class, which is part of the bobcat library (cf. section 24.8.1.5) (the bobcat library is available on systems offering either bisonc++ or flexc++). Bisonc++ can be downloaded from


24.8.3 Bisonc++: using polymorphic semantic values

Instead of using unions to store various semantic values bisonc++ could also use a polymorphic base class to handle semantic values of various types. Using a polymorphic base class is covered in this section. The described method is a direct result of a suggestion initially brought forward by Dallas A. Clement in September 2007.

One may wonder why unions are still used by Bisonc++, as C++ offers inherently superior ways to handle multiple semantic types: a polymorphic base class and a series of derived classes implementing the alternative data types.
On the other hand, a polymorphic base class also seems to imply a lot of additional work: classes must be derived from a base class, virtual members must be declared and overridden in derived classes, and the base class must be aware of the relevant interfaces of all derived classes. All this does more to hinder than to promote the construction of reusable software. So, how to proceed? It turns out that the required effort to implement and use polymorphic semantic values is fairly small. In fact, only a very basic polymorphic semantic base class needs to be implemented. Having defined the polymorphic base class template meta programming techniques can be used to let the compiler create all derived classes we might need. The amount of works turns out to be astonishingly small. What about the 'free lunch'? Well, the approach works fine in situations where we either can deduct the actual semantic value from the grammar (i.e., the syntax) itself, or where we occasionally are willing to use a switch to select the actual semantic value. This rather weak assumption holds true for the grammar used by the program developed in this section, so let's get on with it!

The program developed in this section recognizes input consisting of lines suggesting assignment statements or function calls:

```plaintext
value:
  int
  ident
;

arglist:
  arglist ',' value
|
  value
;

rule:
  ident '(' arglist ')' ';'
|
  ident '=' int ';'
;
```

An essential characteristic of these simple rules is that three different semantic value types are used: int-values, names, and vectors of arguments. Other types could easily have been used as well: doubles, complex numbers, sets; you name it.

Our semantic value must accommodate all these different types, and must also allow us to determine the actual type that's stored in a semantic value in cases where we cannot deduct the actual type merely from the syntax (which happens, e.g., for the different semantic values types contained in an arglist).

In the following sections we'll develop the parser using a polymorphic base class to handle its semantic values. To prevent excessive copying of semantic values the parser's actual semantic value is not the semantic value itself but a `spSemBase`, which is a wrapper around a `std::shared_ptr<SemBase>`, where `SemBase` is our polymorphic base class (cf. section 24.8.3.9). We'll develop the generic `Semantic` class template in steps:

- In the next section we'll start by defining `tags` for the various semantic data types;
- Next, support structs are developed allowing us to indicate whether semantic data can be modified by the parser or not;
- Following this, a trait class is developed allowing us to obtain data types from tags;
Another trait class is needed to determine the data type that is returned by the conversion operators of the different semantic data types;

Hereafter the polymorphic base class `SemBase` is developed;

Finally, the class template `Semantic` is defined, allowing us to define various semantic value classes, all derived from `SemBase`.

The complete demo program is available in the annotation()’s source archive under the directory `yo/concrete/poly2`.

### 24.8.3.1 The parser using a polymorphic semantic value type

In Bisonc++’s grammar specification `%stype` is of course `Semantic`. A simple grammar is defined for this illustrative example. The grammar expects input according to the following rule:

```plaintext
rule:
    IDENTIFIER '(' IDENTIFIER ')' ';' |
    IDENTIFIER '=' INT ';' ;
```

The rule’s actions simply echo the received identifiers and `int` values to `cout`. Here is an example of a decorated production rule, where due to `Semantic`’s overloaded insertion operator the insertion of the object controlled by `Semantic` is automatically performed:

```plaintext
IDENTIFIER '=' INT ';' 
{ 
    cout << $1 << " " << $3 << '\n'; 
}
```

Bisonc++’s parser stores all semantic values on its semantic values stack (irrespective of the number of tokens that were defined in a particular production rule). At any time all semantic values associated with previously recognized tokens are available in an action block. Once the semantic value stack is reduced, the `Semantic` class’s destructor takes care of the proper destruction of the objects controlled by its `shared_ptr` base class.

The scanner must of course be allowed to access the parser’s data member representing the most recent semantic value. This data member is available as the parser’s data member `d_val__`, whose address or reference can be passed to the scanner when it is constructed. E.g., with a scanner expecting an `STYPE__ &` the parser’s constructor could simply be implemented as:

```c++
inline Parser::~Parser()
{
    d_scanner(d_val__)
}
```

### 24.8.3.2 Tagging the actual semantic type: the ‘enum class Tag’

Our program handles three types of semantic values: numbers, text, and vectors of semantic values, which are either numbers or text. These distinct types are indicated by `tag` enumeration values:
enum class Tag { INT, TEXT, VECTOR };

24.8.3.3 (Im)mutable semantic data: two base-structs

In cases where the data stored in the classes derived from the polymorphic base class may either or not be mutable by the parser, there must be a way to indicate so when the derived class is created. Two small support structs define isMutable enum values indicating whether the data should be considered mutable or not. To make matters concrete, let's assume that we'll need INT and TEXT semantic values to be immutable, while VECTOR semantic values need to be mutable. Here are these structs, defined in the anonymous namespace within sembase.h:

```cpp
struct Mutable { enum: bool { isMutable = true }; // BasicTrait, below, from either Mutable or Immutable this trait is associated with a semantic value BasicTrait.
}

struct Immutable { enum: bool { isMutable = false }; }
```

24.8.3.4 Traits of semantic type tags: the ‘TagTrait’ trait class

The TagTrait trait class defines, for each of the distinct semantic values, what the actual data type (DataType) is that is associated with the tag and whether the data are mutable or immutable. Mutable data on the parser's semantic stack may be modified, immutable data may not.

Here is the TagTrait meta template struct template and its specializations for the three data types used by our parser.

```cpp
template <Tag tag>
struct TagTrait;

template <>
struct TagTrait<Tag::INT>: public Immutable {
    typedef int DataType;
};

template <>
struct TagTrait<Tag::TEXT>: public Immutable {
    typedef std::string DataType;
};

template <>
struct TagTrait<Tag::VECTOR>: public Mutable
24.8. USING ‘BISONC++’ AND ‘FLEXC++’

24.8.3.5 Accessing data from derived classes

How to access the data that are actually stored inside a semantic value class that is derived from the semantic values polymorphic base class?

Depending on the status (mutable or immutable) and type (basic or class type) of the actual semantic data we recognize three situations:

- If the data within the semantic value class are mutable, then an accessor should return a reference to the data stored within the semantic value class;
- Immutable non-class type values should be made available by value;
- Immutable class type values should be made available as const references.

Next, a trait class template Trait is defined, requiring a Tag template non-type parameter. This trait class uses the Tag to determine the data type that is associated with the Tag, making its local type DataType as synonym of that data type.

Next, to determine whether an actual data type is a class type or a basic type template meta programming, as outlined in section 23.6.1, is used.

Using template meta programming a value isBasicType of an enum: bool anonymous enum is set to true if DataType represents a basic data type. This enum also defines a value isMutable indicating whether or not the actual data stored in a semantic value class is mutable or not.

Next, conditional to the combinations of isMutable and isBasicType the Trait trait class defines the type ReturnType. For this the available std::conditional trait class is used (cf. section 23.6.2).

Now we’re able, e.g., to state Trait<Tag::INT>::DataType to obtain the int data type, or to state Trait<Tag::VECTOR>::ReturnType to obtain the 'std::vector<std::shared_ptr<SemBase> &' return type.

Here is the implementation of the trait class template Trait:

```cpp
struct C2 {
  char _[2];
};

struct C1 {
  char _;
};

template <typename T>
C1 test(...) {
}

template <typename T>
C2 test(void (*T::*)) {
}
```
template <Tag tg_>
struct Trait: public TagTrait<tg_>
{
    typedef typename Trait<tg_>::DataType DataType;
    enum: bool
    {
        isMutable = Trait<tg_>::isMutable,
        isBasicType = sizeof(test<DataType>(0)) == sizeof(C1)
    }

    typedef typename
    std::conditional<
        isMutable,
        DataType &,
        typename std::conditional<
            isBasicType,
            DataType,
            DataType const &
        >::type
    >::type ReturnType;
};

24.8.3.6 The polymorphic base class ‘SemBase’

The implementation of the polymorphic semantic value in fact implements a type-safe polymorphic semantic union. In C++ this data type does not exist, but the parser may handle semantic values as unions. It always knows the actual type of semantic value that’s used at a particular point in the grammar. If not, it can inspect SemBase’s tag fields.

The parser, knowing what the actual semantic type is, can always perform a downcast, resulting in a very lightweight SemBase base class.

The SemBase class only requires one virtual member: an empty virtual destructor. Downcasting is encapsulated in SemBase’s ‘as’ member. Consequently, the parser, defining its semantic value as a spSembase (cf. section 24.8.3), can use constructions like

$1->as<Tag::STRING>()$

to access the std::string that is stored in the actual semantic value to which $1$ points. Here is SemBase’s class interface (the implementation of its ‘as’ member template follows, implementations of its remaining members are trivial):

class SemBase
{
    Tag d_tag;

    public:
        SemBase(SemBase const &other) = delete;
    virtual ~SemBase();
    Tag tag() const;

    template <Tag tg_>
24.8. USING 'BISONC++' AND 'FLEXC++'

```cpp
typename Trait<tg_>::ReturnType as() const;
protected:
    SemBase(Tag tag);
};
```

### 24.8.3.7 The class template ‘Semantic’, derived from ‘SemBase’

Finally we reach the class template `Semantic`, publicly derived from `SemBase`. This class template only requires one `Tag` non-type parameter, defining the kind of data that must be handled by objects of the class.

Like `SemBase` the class `Semantic` is extremely light-weight. Only initialization and conversion to the encapsulated data member need to be supported.

It always defines its data member as mutable, which is fine since the class itself doesn’t manipulate the data and proper access to the data is guaranteed by its `operator ReturnType() const` conversion operator. `ReturnType`, of course, is obtained through the `Trait` trait class template.

Here is the class’s interface and member definitions:

```cpp
template <Tag tg_>
class Semantic: public SemBase
{
    typedef typename Trait<tg_>::DataType DataType;
    enum: bool { isMutable = Trait<tg_>::isMutable };

    mutable DataType d_data;

    public:
    typedef typename Trait<tg_>::ReturnType ReturnType;
    Semantic(DataType const &data);
    operator ReturnType() const;
};
template <Tag tg_>
Semantic<tg_>::Semantic(DataType const &data)
:
    SemBase(tg_),
    d_data(data)
{};

template <Tag tg_>
Semantic<tg_>::operator ReturnType() const
{
    return d_data;
}
```

To access the information stored in a semantic value class the `SemBase::as` member template is used. This member template is provided with a tag value and returns the value stored inside the actual `Semantic` object. It’s use was shown in the previous section. Here is its implementation:

```cpp
template <Tag tg_>
```
CHAPTER 24. CONCRETE EXAMPLES

inline typename Trait<tg_>::ReturnType SemBase::as() const
{
    return dynamic_cast<Semantic<tg_> const &>(*this);
}

24.8.3.8 Adding new semantic data types

These are the steps to take when another semantic data type must be added to an existing set:

- Add a tag name representing the new semantic value type to the enum class Tag (section 24.8.3.2);
- Add a specialization to the TagTrait trait class (section 24.8.3.4) defining the added semantic value's data type, and whether or not the data on the parser's semantic stack should be considered (im)mutable.

24.8.3.9 The parser's semantic value: 'spSemBase'

The parser uses spSemBase as its semantic value. The class spSemBase is a wrapper around std::shared_ptr<SemBase>, offering a constructor member template which must be given a pointer to a dynamically allocated Semantic object. Its interface is all that is required:

```c++
#ifndef INCLUDED_SPSEMBASE_
#define INCLUDED_SPSEMBASE_
#include "../sembase/sembase.h"

class spSemBase: public std::shared_ptr<SemBase>
{
    public:
        spSemBase() = default;

        template <typename Type>
        spSemBase(Type *obj);
    
    template <typename Type>
    inline spSemBase::spSemBase(Type *obj)
    {
        std::shared_ptr<SemBase>{obj}
    }

    #endif
```

24.8.3.10 The parser specification file

Now that the Semantic class template has been developed it's time to put it to use in the Parser class. The parser's semantic value is spSemBase. The parser's base class must be informed about this, for which the %baseclass-preinclude directive should be used. All other directives are standard and do not require further explanations:
%filenames parser
%scanner     ../scanner/scanner.h
%baseclass-preinclude ../spsembase/spsembase.h
%stype       spSemBase

%token INT IDENT

The grammar’s rules simply consist of a series of rule nonterminals:

```
rules:
  rules rule
  |  rule
```

Int values are stored in Semantic<Tag::INT> objects, text is stored in Semantic<Tag::TEXT> values:

```
int:
  INT
  {$$ = new Semantic<Tag::INT>(A2x(d_scanner.matched()).to<int>())};

ident:
  IDENT
  {$$ = new Semantic<Tag::TEXT>(d_scanner.matched());}
```

Comma separated lists of arguments are processed as follows: the first argument is stored in a Semantic<Tag::VECTOR>; additional values are added to the vector using vector::push_back. Note that we’ve defined the vector as mutable: addition of values to the vector is OK, but the values themselves remain as-is, and so the vector stores shared pointers to SemBase const values:

```
value:
  int
  |  ident

arglist:
  arglist ',' value
  {$1->as<Tag::VECTOR>().push_back($3);}
  |  value
  {$$ = new Semantic<Tag::VECTOR>{}
      vector<shared_ptr<SemBase const>> {($1)};}
```
The definition of the rule production rule completes our little grammar: an alternative suggesting an assignment echoes the received names and values, and an alternative suggesting a function call uses a support member display to display the received name and arguments:

```plaintext
rule:
  ident '(' arglist ')' ';'
  \{ display($1, $3); \}
| ident '=' int ';'
  \{ cout << $1->as<Tag::TEXT>() << " = " << $3->as<Tag::INT>() << ";\n"; \}
;
```

### 24.8.3.11 The scanner using a polymorphic semantic value type

The scanner for the polymorphic parser is simple and only needs to recognize numbers, identifiers and some simple characters, returned as character tokens. Here is the scanner's complete specification file, as used by flexc++:

```plaintext
%filenames scanner
%
[[:space:]]+    // skip white space
[[:digit:]]+     return Parser::INT;
[[:alpha:]]_[[:alnum:]]_+ return Parser::IDENT;
.
    return matched()[0];
```
Index

!=, 294
-std=c++11, 8, 9
->, 434
>*, 434
-O0, 166
-lpthread, 610
-pthread, 610
., 434
..., 753
.h, 19
.ih extension, 159
.template, 821
//, 13
::, 25, 276
::template, 821
<, 294
<=, 294
= 0, 388
= default, 144, 212
= delete, 144
==, 294
>, 294
>=, 294
>>, 261
>>: with templates, 318
[begin, end), 300
[first, beyond), 302, 306, 316, 330
[first, last), 526
[left, right), 464
#define __cplusplus, 17
#error, 708
#ifdef, 19
#ifndef, 19
#include, 935
%baseclass-preinclude, 940
%debug, 935
%filenames, 936, 940
%left, 941
%nonassoc, 941
%option c++, 933
%prec, 941
%right, 941
%scanner, 940
%stype, 940
%token, 941
%type, 941
%union, 940
&, 38
__cplusplus, 17, 18

abt exception, 249
abs, 347
absolute position, 409
abstract base class, 422, 896
abstract classes, 387
access, 50
access files, 103, 110
access promotion, 362
access rights, 124
accessor, 124, 127, 262
accumulate, 527
actions, 938
adaptor: inserter, 467
adaptor: object to iterator, 463
adaptor: required typedefs, 467
add_const, 848
add_cv, 848
add_lvalue_reference, 848
add_pointer, 848
add_rvalue_reference, 848
add_volatile, 848
addExceptFd, 908
addReadFd, 908
address, 297
address of objects, 194
address-of operator, 38
addWriteFd, 908
adjacent_difference, 527
adjacent_find, 528
adjustfield, 96
Aho, A.V., 304
Alexandrescu, A., 721, 815, 822, 859, 869, 870
Alexandrescu, H., 856
algorithm: header file, 525
allocate, 297
allocate arrays, 177
allocate memory, 277
allocate objects, 176
allocate primitive types, 176
allocator, 297, 834
allocator class, 453
allocator_arg_t, 666
ambiguity, 133, 392, 393
amd, 51
and, 290
and_eq, 290
angle bracket notation, 295, 296, 453
angle brackets, 707
angle brackets: consecutive, 318
anonymize, 207
anonymous, 457, 469
anonymous function object, 487
anonymous object, 147, 286
anonymous object: lifetime, 148
anonymous pair, 296
anonymous type, 687, 688
anonymous variable, 39
anonymous variable: generic form, 296
ANSI/ISO, 13, 17, 48, 57, 86, 89, 93
app, 104, 116
append, 75
arg, 347
argument_type, 606, 842
arithmetic function object, 455
arithmetic operations, 455, 919
arithmetic type, 847
array, 298
array bounds, 298, 300
array bounds overflow, 257
array constructors, 298
array-to-pointer transformation, 690
array: 0-sized, 178
array: dynamic, 177
array: enlarge, 178
array: expandable, 300
array: fixed size, 178, 179, 298
array: header file, 298
array: local, 178
array: member functions, 299
as (SemBase::as), 953
ASCII, 101, 107, 108, 312
ascii-value, 891
assembly language, 10
assign, 75, 300, 305, 315
assignment, 373
assignment: pointer to members, 433
assignment: refused, 374
assignment: sequential, 194
associative array, 317, 326, 335
async, 660, 661
asynchronous alarm, 905
asynchronous input, 905
asynchronous provider, 653
asynchronous return object, 653
at, 75, 299, 301, 315, 320, 339
ate, 104, 117
atoi, 111
atomic actions, 629
atomic data types, 646
atomic: header file, 645
atomic<Trivial>, 646
atomic<Type>, 646
atomic_compare_exchange_strong(_explicit), 649
atomic_compare_exchange_weak(_explicit), 649
atomic_exchange(_explicit), 649
atomic_fetch_add(_explicit), 649
atomic_fetch_and, 649
atomic_fetch_or, 649
atomic_fetch_sub(_explicit), 649
atomic_fetch_xor, 649
atomic_init, 649
atomic_is_lock_free, 649
atomic_load(_explicit), 649
atomic_store(_explicit), 649
atto, 611
Austin, T., 644
auto, 44
auto_ptr, 477
auto_ptr: deprecated, 472
automatic expansion, 300
available member functions, 375
avoid global variables, 20
b, 49
back, 75, 299, 301, 305, 311, 315
back_inserter, 467
backdoors, 127
background process, 900
bad, 91
bad_alloc, 188, 236, 243, 280
bad_cast, 236, 401
bad_exception, 234, 236
bad_function_call, 713
bad_typeid, 236, 404
badbit, 90
base class, 355, 374, 886, 896
base class destructor, 366
base class initializer, 363
base class initializers: calling order, 372
base class: converting to derived class, 399
base class: hiding members, 367
base class: prototype, 406
INDEX

base class: virtual, 393
bash, 114
BASIC, 10
basic guarantee, 242
basic operators of containers, 294
basic_, 85
basic_ios.h, 89
basic_regex, 495
beg, 102, 110, 414
begin, 75, 144, 299, 301, 305, 315, 320, 329, 339, 464
bernoulli_distribution, 510
bidirectional_iterator_tag, 803
BidirectionalIterator, 466, 803
bigraphs, 3
binary, 104, 117
binary constant, 49
binary constants, 48
binary file, 101, 108, 117, 118
binary function object, 460
binary input, 107
binary operator, 919
binary output, 96, 101
binary predicate, 462
binary tree, 600
binary_function, 919
binary_search, 530
bind1st, 460
bind2nd, 460
bind2nd: perfect forwarding, 760
binder, 460
binders.h, 759
binomial_distribution<IntType = int>, 510
bison, 933, 938, 939
bison++, 933, 938
bison++: grammar file, 939
bit-flags, 48
bitand, 290
bitor, 290
bits/stl_function.h, 919
bitwise, 919
bitwise and, 93, 919
bitwise operations, 919
bobcat, 938, 947
Bobcat library, 238, 492, 879, 883
bool, 48, 322, 330
boolalpha, 97
bound friend, 770, 772, 800
bridge design pattern, 405
broken.promise, 656
bucket, 339
bucket number, 335
bucket_count, 339
bucket_size, 339
buffer, 409
buffer overflow, 33
built-in, 48
C++11, 141, 318, 335, 492, 685
C++: function prevalence rule, 684
c_str, 77
call back, 172
call_once, 630
calling order of base class initializers, 372
calloc, 175
candidate functions, 715
capacity, 76, 301
case-insensitive, 70
catch, 218, 229
catch: empty enum, 450
caucho_distribution<RealType = double>, 511
cbegin, 76, 299, 301, 315, 320, 339
ccbuild, 9
cend, 76, 299, 301, 315, 320, 339
centi, 611
cerr, 29, 100, 113
chain of command, 407
char, 85
char *, 262
chi_squared_distribution<RealType = double>, 512
child process, 895, 897
chrono namespace, 612
chrono: header file, 611, 612, 614, 615
cin, 29, 89, 107, 113
circular queue, 722
class, 12, 30, 57, 450, 681, 836
class hierarchy, 355, 385
class interface, 770
class name, 403
class scope, 432
class template, 692, 721, 722, 919
class template: adding functionality, 780
class template: as base class, 838
class template: conditional data organization, 826
class template: declaration, 735, 767
class template: deducing parameters, 767
class template: default argument, 735
class template: default arguments, 734
class template: defining a type, 823
class template: derivation, 780
class template: derived from ordinary class, 785
class template: friend function template, 720
class template: generate types, 722
class template: identifying subtypes, 739
class template: instantiation, 767
class template: member instantiation, 767
class template: member template, 726
class template: nested, 799
class template: non-type parameter, 724
class template: partial specialization, 745, 748
class template: partially compiled, 780
class template: pointer to, 767
class template: reference to, 767
class template: shadowing parameters, 728
class template: static members, 737
class template: subtype vs. static members, 817
class template: transformation to a base class, 692
class template: using friend, 769
class template: values without objects, 824
class template: wrapped in simpler template, 870
class type, 847
class vs. typename, 681
Class(T&) cannot be overloaded with Class(T const&), 759
class-type parameters, 158
class-type return values, 158
class: abstract, 387
class: concept defined, 427
class: enforcing constraints, 833
class: externally declared functions, 427
class: having pointers, 484
class: local, 153, 376
class: move-aware, 202
class: ordinary, 692
class: policy, 833
class: trait, 841
class: without data members, 388
clear, 93, 301, 306, 315, 320, 329, 339
climits: header file, 709
Cline, 28
clock, 614
clock: now, 614
clog, 100
close, 103, 111, 415
closure object, 487
closure type, 487
code, 658
code bloat, 790, 793, 839
Coetmeur, A., 938
collating order, 404
collision, 336
command language, 905
common data fields, 163
common pool, 176
compare, 76
compare_exchange_strong, 647
compare_exchange_weak, 647
compile-time, 16, 381, 383, 397, 679, 708
compiler, 7–9
compiler firewall, 405
compiler flag, 8
compiler option, 9
compiler option: -pthread, 610
compl, 290
complex, 345
complex numbers, 345
complex: header file, 345
composition, 136, 158, 355, 370
compound type, 847
condition flags, 90
condition member functions, 91
condition state, 90
condition_variable, 639, 640
condition_variable: header file, 638, 640, 642
condition_variable_any, 639, 642
conditional, 848
conj, 347
consecutive closing angle brackets, 318
cost, 26, 691
const data and containers, 295
const function attribute, 14
const functions, 27
cost member, 127
cost member functions, 145, 388
const-qualification, 146
const: generalized expression, 167
const_cast<type>(expression), 54
const_iterator, 299, 301, 315, 320, 339, 464
const_pointer_cast, 482
const_reverse_iterator, 299, 301, 315
constant expression, 683
constant-expression constructor, 169
constant-expression function, 167
costexpr, 167, 631
construct, 297
construction: delegate to base classes, 365
constructor, 286, 372, 454
constructor notation, 52
constructor: and exceptions, 249
constructor: calling order, 366
constructor: default, 125, 126
constructor: delegation, 141
constructor: of derived classes, 363
constructor: primitive type, 684
container, 293
container without angle brackets, 295
container: basic operators, 294
container: data type requirements, 294
container: equality tests, 294
container: initialization, 298
container: nested, 318
INDEX

container: ordering, 294
container: storing pointers, 295
container: storing const data, 295
conversion operator, 263
conversion operator: explicit, 267
conversion rules, 50
conversions, 725
conversions: binary to text, 106
conversions: text to binary, 111
copy, 76, 531, 784
copy construction, 144
copy constructor, 149, 196, 364
copy constructor: suppressed, 205
copy elision, 213
copy non-involved data, 304
copy_backward, 532
copyfmt, 94
cos, 347
cosh, 347
count, 320, 326, 329, 331, 339, 342, 344, 532, 613
count_if, 460, 533
coupling, 12
cout, 29, 89, 100, 113, 899
crbegin, 76, 299, 301, 315, 320
create files, 103
cref(arg), 687
crend, 77, 299, 301, 315
cstddef, 51
cstdint, 52
cstdio, 14
cstdlib: header file, 769
cumulative distribution function, 507
cur, 102, 110, 414
curiously recurring template pattern, 790, 792
current_exception, 672
cv_status, 640
Cygnus, 9
Cygwin, 9
daemon, 899, 900, 914, 916
data, 77, 299, 301
data hiding, 10, 12, 32, 124, 127, 166, 358, 361, 409, 427
data integrity, 427
data member, 124, 409
data member initializers, 139
data member: initialization, 141
Data Structures and Algorithms, 304
data.cc, 164
database applications, 109
deallocate member, 297
deallocate memory, 277
Debian, 8
dec, 96
deca, 611
deci, 611
decimal constants, 48
declaration section, 940
declarative region, 57
declare iostream classes, 86
dec_type, 45, 685
decrement operator, 267
default, 144
default arguments, 15
default constructor, 177, 196, 294, 363
default implementation, 412
default value, 302, 308, 316
default_error_condition, 240
deferred, 661
define members of namespaces, 66
deleagating constructors, 141
delete, 144, 175, 176, 276
deleter, 473, 475, 479
delete[], 178, 185
deletions, 304
delimiter, 470
denorm_min, 709
denorm_style, 710
dequeue, 314, 463
deequeue constructors, 314
dequeue: header file, 314
derivation, 355
derived class, 355, 374, 399
derived class destructor, 366
derived class vs. base class size, 376
derived class: using declaration, 368
design pattern, 12, 388, 895
design pattern: Prototype, 422
design pattern: template method, 896
Design Patterns, 239, 895
design patterns, 239
destroy, 297
destructor, 125, 183, 386
destructor: and exceptions, 255
destructor: called at exit, 898
destructor: calling order, 366
destructor: derived class, 366
destructor: empty, 387
destructor: explicit call, 182, 184
destructor: for policy classes, 839
destructor: inline, 387
destructor: main task, 183
destructor: virtual, 387
detach, 622
device, 88, 877
digits, 710
digits10, 710
direct base class, 358
display field width, 95
display floating point numbers, 95, 98
distance, 466
divides, 458
domain_error, 236
DOS, 117
double free, 482
double initialization, 375
doubly ended queue, 314
down-casting, 399
downcast, 952
dup, 899
dup2, 899, 903
duration, 614
duration template, 612
dynamic arrays, 177
dynamic binding, 383
dynamic cast, 399
dynamic growth, 304
dynamic polymorphism, 721, 790, 839
dynamic_cast, 399, 402
dynamic_pointer_cast, 482
dynamically allocated variables, 725
E, 49
eyearly binding, 381, 383
eback, 409, 881, 884, 889
ECHO, 936
ECMAScript, 496
efficiency, 336
egptr, 879
epsilon, 710
equality operator, 294
equivalent, 240
erase, 77, 301, 306, 316, 321, 326, 330, 331, 340, 344
erro, 939
erro, 239
error, 939
error code, 217
error_category, 237, 239
error_code, 237, 238, 658
escape sequence, 48
exa, 611
exceptFd, 907
exception, 93, 400
Exception (Bobcat), 238
exception class, 235, 236
exception guarantees, 242
exception handler: order, 229
exception neutral, 244
exception safe, 241
exception specification list, 232
exception: and constructors, 249
exception: and destructors, 255
exception: and new, 243
exception: and new[], 280
exception: and streambuf, 407
exception: bad_alloc, 188
exception: header file, 236
exception: replace, 249
exception: standard, 236
exception_ptr, 672
Exceptional C++, 244
exceptions, 217
exceptions (function), 235
exchange, 647
exec...., 898
exit, 217, 223, 898, 900
exit status, 897
exit: avoid, 183
exit: calling destructors, 898
exp, 347
expandable array, 300
expected constructor, destructor, or type conversion, 817
explicit, 266
explicit instantiation declaration, 696
explicit template type arguments, 699
explicit type specification, 689
explicit: conversion operator, 267
exponential_distribution < RealType = double>, 513
exponentiation, 49
export, 56
expression: actual type, 399
expression: type of, 403
extended friend, 429, 779
testable literals, 289
extern, 8, 767
extern C, 18
extern C\hyperpage, 18
extern template, 735–737
extracting strings, 107
extraction operator, 29, 30, 88, 106, 107
extreme_value_distribution < RealType = double>, 513
F, 49
factory function, 133
factory functions, 210
fail, 91, 102
failbit, 90, 110
failure class, 235
false, 49, 553, 562
femto, 611
fetch_\^=, 648
fetch_\|=, 648
fetch_add, 647
fetch_and, 648
fetch_sub, 647
field ‘...’ has incomplete type, 772
field selector, 434
field width, 284
FIFO, 293, 310
FILE, 85
file descriptor, 103, 114, 886, 888
file descriptors, 88, 877, 901
file flags, 104
file is rewritten, 105
file modes, 104
file rewriting: preventing, 104
file: binary mode, 104
file: copying, 112
filebuf, 88, 414
fill, 94, 299, 537
fill characters, 96
fill_n, 537
FILO, 293, 333
final, 390
find, 77, 321, 327, 330, 332, 340, 342, 345, 538
find_end, 539
find_first_of, 77, 540
find_if, 542
find_last_not_of, 78
find_last_of, 78
first, 296
first in, first out, 293, 310
first in, last out, 293, 333
first argument type, 606, 842
fisher_f_distribution < RealType = double>, 514
Fistream, 891
fixed, 98
fixed-size array, 298
flag_type, 496
flags, 94
flags: of ios objects, 93
flex, 933
flexc++, 934–936, 938, 947, 956
flexc++: debugging code, 935
flexc++: setDebug, 935
float_denorm_style, 710
float_round_style, 712
floatfield, 98
flow-breaking methods, 217
flush, 99, 102
fopen, 100, 107
for range declaration, 47
for-statement, 47
for_each, 543, 915
for_each: vs. transform, 596
fork, 609, 895, 896, 900
formal type, 679, 681, 723
format, 501
format specifier, 501
format string, 501
format_default, 501
format_first_only, 501
format_no_copy, 501
format_sed, 501
formatted input, 107
formatted output, 96, 101
formatting, 90, 94
formatting commands, 88
formatting flags, 93, 96
forward, 756
forward class reference, 157
forward declaration, 85, 158, 447
forward declaration: enum class, 42
forward iterators, 73
greater_equal, 458

hardware_concurrency, 624
has_denorm, 710
has_denorm_loss, 710
has_infinity, 710
has_nothrow_assign, 848
has_nothrow_copy_constructor, 848
has_nothrow_default_constructor, 848
has_nothrow_destructor, 848
has_quiet_NaN, 710
has_signaling_NaN, 710
has_trivial_assign, 848
has_trivial_copy_constructor, 848
has_trivial_destructor, 848
has_virtual_destructor, 848
hash_function, 335
hash_value, 335
hash_function, 340
hashing, 335
header file, 57, 88, 155, 160
header section, 940
heap, 600
hecto, 611
hex, 97
hexadecimal constants, 48
hidden data member, 418
hiding: base class members, 367
hierarchic sort, 782
hierarchy of code, 355
high_resolution_clock, 615
Hopcroft J.E., 304
hours, 612
http://bisoncpp.sourceforge.net/, 947
http://bobcat.sourceforge.net, 238, 492, 879, 883
http://bobcat.sourceforge.net/, 938
http://en.wikipedia.org/wiki/C++11, 6
http://flexcpp.sourceforge.net/, 938
http://gcc.gnu.org, 9
http://oreilly.com/catalog/, 609
http://publications.gbdirect.co.uk/c_book/, 3
http://sourceforge.net/projects/cppannotations/, i
http://sources.redhat.com, 9
http://www.debian.org, 8
http://www.gnu.org, 8
http://www.gnu.org/licenses/, 3
http://www.linux.org, 8
http://www.oreilly.com/catalog/lex, 933
http://www.research.att.com/.../27
http://www.sgi.com/.../STL, 295
http://www.trolltech.com, 125
http://www.parashift.com/c++-faq-lite/, 28
http://yodl.sourceforge.net, 3
human-readable, 96
I/O, 85
I/O library, 85
I/O multiplexing, 905
icase, 496
icmake, 9
id, 618, 623
identically named member functions, 372
identifier: initial underscore, 289
identifiers starting with an underscore, 56
IEC-559, 710
IEEE-754, 710
IFdNStreambuf, 882
IFdSeek, 885
IFdStreambuf, 881, 913
ifstream, 107, 110, 122
ifstream constructors, 110
ignore, 108
imag, 347
imaginary part, 346, 347
implementation, 123
implementation dependent, 427
implemented-in-terms-of, 381
implicit conversion, 374
in, 104, 105, 116
in_avail, 407
INCLUDE, 156, 159
include guard, 19
includes, 547
increment operator, 267
index operator, 257, 298, 300, 315, 320, 326
indirect base class, 358
inequality operator, 294
infinity, 710
inheritance, 154, 355, 357
inheritance: access to base class member, 362
inheritance: multiple, 370
inheritance: no derived class constructors, 365
inheritance: private derivation, 781
init, 370, 899, 900
initialization, 141, 176, 196, 298
initialization: static data member, 164
initializer list, 43, 141
initializer_list, 143
initializer_list: header file, 43, 143
inline, 150–152, 284, 387, 454
inline function, 152
inline member functions, 443
inline: avoid!, 153
inline: disadvantage, 153
inline: static members, 171
inner_product, 549
inplace_merge, 550
input, 106, 113
input language, 938
input_iterator_tag, 802
InputIterator, 465, 802
InputIterator1, 465
InputIterator2, 465
inserter, 467
inserter: and non-STL classes, 468
inserter: required typedefs, 467
inserting streambuf *, 113
insertion operator , 29, 86, 100, 101, 260
insertions, 304
instantiation, 679, 681
instantiation declaration, 696, 737
instantiation function, 736
int32_t, 52
INT_MAX, 709
integer division, 53
integral conversions, 725
integral type, 847
interface, 123, 387
interface functions, 126
internal, 96
internal buffer , 102
internal header , 159
internal header file, 898
invalid_argument, 236
iomanip, 94
iomanip: header file, 89
ios, 86, 89, 113, 448, 695, 885
ios object: as bool value, 92
ios: header file, 88, 886
ios::exceptions, 235
ios::fail, 103, 111
ios_base, 86, 89
ios_base.h, 89
iosfwd, 70, 88
iosfwd: header file, 86
iostate, 235
iostream, 19, 30, 103, 110
iostream.h, 19
iostream: header file, 89, 101, 107
is-a, 381, 405, 406
is-implemented-in-terms-of, 405
is_abstract, 849
is_arithmetic, 849
is_array, 849
is_assignable, 849
is_base_of, 849
is_bounded, 710
is_class, 849
is_compound, 849
is_const, 849
is_constructible, 849
is_convertible, 849
is_copy_assignable, 849
is_copy_constructible, 849
is_default_constructible, 849
is_destructible, 849
is_empty, 849
is_enum, 849
is_exact, 710
is_floating_point, 849
is_function, 849
is_fundamental, 849
is_iec559, 710
is_integer, 711
is_integral, 849
is_literal_type, 849
is_lock_free, 647
is_lvalue_reference, 849
is_member_function_pointer, 850
is_member_object_pointer, 850
is_member_pointer, 850
is_modulo, 711
is_move_assignable, 850
is_move_constructible, 850
is_nothrow..., type trait, 847
is_nothrow_assignable, 850, 853
is_nothrow_constructible, 850, 853
is_nothrow_copy_assignable, 853
is_nothrow_copy_constructible, 850, 853
is_nothrow_default_constructible, 850, 853
is_nothrow_destructible, 850, 853
is_nothrow_move_assignable, 850, 853
is_nothrow_move_constructible, 850, 853
is_object, 850
is_open, 104, 111, 415
is_pod, 850
is_pointer, 850
is_polymorphic, 850
is_reference, 850
is_rvalue_reference, 850
is_same, 850
is_scalar, 851
is_signed, 711, 851
is_specialized, 711
is_standard_layout, 851
is_trivial, 851
is_trivially_assignable, 851
is_trivially_constructible, 851
is_trivially_copy_assignable, 851
is_trivially_copy_constructible, 851
is_trivially_copyable, 851
is_trivially_default_constructible, 851
is_trivially_default_destructible, 851
is_trivially_move_assignable, 851
lock_guard, 632
log, 347
logic_error, 236
logical function object, 459
logical operations, 919
logical operators, 459
logical_and, 459
logical_not, 459
logical_or, 459
lognormal_distribution<RealType = double>, 517
long double, 48
long long, 48
long long int, 51
longjmp, 217, 220
lower_bound, 323, 330, 554
lowest, 711
lsearch, 282
lseek, 886
LSP, 381, 405
Ludlum, 60
lvalue, 39, 257, 267, 466, 476
lvalue reference, 39
lvalue transformations, 690, 725
lvalue-to-rvalue transformation, 690
lvalue: distinguish from rvalue, 927
macro, 16, 166, 286
main, 13, 14
make, 9
make_exception_ptr, 672
make_heap, 601
make_shared, 484
make_signed, 851
make_unsigned, 851
malloc, 175, 176, 180, 185, 189
manipulator, 88, 125, 284
manipulator: as objects, 286
map, 317
map constructors, 318
map: header file, 317, 326
marked sub-expression, 493, 498
Marshall Cline, 28
match_any, 501
match_continuous, 501
match_default, 501
match_flag_type, 501
match_not_bol, 501
match_not_bow, 501
match_not_eol, 501
match_not_eow, 501
match_not_null, 502
match_prev_avail, 502
match_results, 498
matched, 941
mathematical functions, 347
max, 508, 556, 613, 616, 711
max_heap, 526, 602
max_bucket_count, 341
max_element, 556
max_exponent, 711
max_exponent10, 711
max_load_factor, 341
max_size, 79, 294, 297, 302, 306, 316, 324, 330, 341
mega, 611
member function, 31, 69, 123, 305, 311, 409, 476, 481
member function: available, 375
member function: called explicitly, 368
member function: identically named, 372
member function: overloading, 15
member function: pure virtual implementation, 388
member functions, 313
member initializer, 136
member template, 726
member: class as member, 441
member: const, 127
member: in-class, 151
member: overriding, 386
memcpy, 201, 246
memory allocation, 175
memory buffers, 86
memory consumption, 418
memory leak, 175, 176, 178, 186, 226, 229, 295, 387, 471, 484
memory: header file, 180, 471, 478, 484, 834
memory: initialization, 177
memory_order, 648
merge, 306, 557
merging, 526
message, 240
micro, 611
microseconds, 612
milli, 611
milliseconds, 612
min, 508, 559, 613, 616, 711
min_element, 560
min_exponent, 711
min_exponent10, 711
mini scanner, 935
minus, 457
minutes, 612
mixing C and C++ I/O, 88
modifier, 261
modulus, 458
move, 476
move assignment, 208
O0, 166
object, 22
object oriented approach, 11
object: address, 194
object: allocation, 176
object: anonymous, 147
object: assign, 190
object: parameter, 197
object: static/global, 126
obsolete binding, 20
oct, 97
octal constants, 48
off_type, 102, 109
ofstream, 100, 103, 122
ofstream constructors, 103
once_flag, 630
one definition rule, 123, 695
open, 103, 110, 415, 879
openmode, 104, 415
operator, 193
operator and, 290
operator and_eq, 290
operator bitand, 290
operator bitor, 290
operator bool, 476, 480
operator compl, 290
operator delete, 180, 276
operator delete[], 278
operator new, 180, 236, 274
operator new(sizeInBytes), 177
operator new[], 277
operator not, 290
operator not_eq, 290
operator or, 290
operator or_eq, 290
operator overloading, 192, 257
operator overloading: within classes only, 291
operator xor, 290
operator xor_eq, 290
operator!=, 282, 458, 465
operator(), 282, 284, 571
operator*, 347, 457, 465, 476, 480
operator*=: 347
operator+, 269, 346, 456, 457, 527
operator++, 267, 465, 648
operator+=, 347, 648
operator-, 347, 457
operator --, 267, 648
operator-=, 347, 648
operator->, 476, 481
operator/!, 347, 458
operator/=, 347
operator: keywords, 56
operator<, 317, 335, 458, 557, 558, 562, 565, 566, 569, 584, 586–589, 591, 599, 602, 603
operator<<, 347
operator< =, 458
operator=, 476, 480
operator==, 337, 458, 465, 582, 583, 597, 598
operator>, 458
operator>=, 458
operator>>, 107, 261, 347
operator%, 458
operator&, 35, 38, 919
operator&&, 648
operator&&, 459
operator^=, 648
operator~-, 919
operator!=, 648
operator||, 459
operators of containers, 294
operators: textual alternatives, 290
operator[], 257, 262, 833, 927
options, 935
or, 290
or_eq, 290
ordered pair, 347
ordinary class, 679, 692
ordinary function, 679
ostream, 86, 89, 90, 100, 101, 122, 285, 286, 388
ostream constructor, 100
ostream coupling, 113
ostream: define using 0-pointer, 100, 107
ostream: header file, 89, 101
ostream_iterator, 470
ostreambuf_iterator, 470
ostringstream, 86, 100
out, 105, 116
out of memory, 189
out-of-line functions, 153
out_of_range, 236
output, 113
output formatting, 86, 89
output_iterator_tag, 802
OutputIterator, 466, 802
overflow, 408, 412, 877, 880
overflow_error, 237
overloadable operators, 290
overloaded assignment, 294
overloading: by const attribute, 15
overloading: function template, 700
overpopulation, 609, 650
override, 383, 390
overriding members, 386
overview of generic algorithms, 295
owns_lock, 635
private backdoor, 260
private derivation, 372
private derivation: too restrictive, 362
private inheritance, 405
private members, 770
probability density function, 507
problem analysis, 355
procedural approach, 11
process ID, 896
process id, 896
profiler, 152
Prolog, 10
promise, 668
promise_already_satisfied, 657
promotion, 266
promotions, 725
protected, 32, 360, 881
protected derivation: too restrictive, 362
protocol, 387
Prototype design pattern, 422
prototyping, 8
Proxy Design Pattern, 927
Proxy: stream insertion and extraction, 928
pthreads, 609
Pthreads Programming, 609
ptr_diff, 606
public, 32, 166, 372
pubseekoff, 408, 414
pubseekpos, 409
pubsetbuf, 409
pubsync, 408
pure virtual functions, 388
pure virtual member: implementation, 388
push, 311, 314, 335
push_back, 79, 302, 307, 316, 467
push_front, 79, 307, 316, 467
push_heap, 602
put, 101
putback, 109, 887
qsort, 769
qsort(), 172
Qt, 125
qualification conversions, 725
qualification transformation, 691
queue, 310
queue data structure, 310
queue: header file, 310, 312
quiet_NaN, 711
radix, 93, 711
rand, 507
random, 304
random access, 466
random number generator, 571
random: header file, 507, 508
random_access_iterator_tag, 803
random_shuffle, 570
RandomAccessIterator, 466, 803, 805
RandomIterator, 930
range, 47
range of values, 300
range-based for-loop, 47, 925
range_error, 237
Ranger, 925
ratio template, 611
ratio: den, 611
ratio: header file, 611, 612
ratio: num, 611
raw memory, 177, 180
raw string literal, 48
rbegin, 79, 299, 302, 307, 316, 324, 330, 464, 809
rdbuf, 90, 114, 899
rdstate, 93
read, 109
read first, test later, 113
readFd, 907
reading and writing, 88
readsome, 109
real, 347
real part, 346, 347
realloc, 189
recompilation, 358
recursive_mutex, 629
recursive_timed_mutex, 629
redirection, 114, 899
ref(arg), 687
reference, 35, 285, 374
reference operator, 35
reference parameter, 138
reference wrapper, 687
reference: to class template, 767
regcomp, 492
regex, 495
regex: header file, 492, 495, 498, 503
regex_constants, 496
regex_match, 503, 504
regex_replace, 503, 505
regex_search, 503, 504
regex_replace, 492
regular expression, 492, 935
rehash, 341
reinterpret_cast, 822
reinterpret_to_smaller_cast, 822
relational function object, 458
relational operations, 919
relationship between code and data, 355
relative address, 433
release, 477, 635
remove, 307, 572
remove_all_extents, 851
remove_const, 851
remove_copy, 573
remove_copy_if, 574
remove_cv, 851
remove_extent, 851
remove_if, 308, 575
remove_pointer, 852
remove_reference, 852
remove_volatile, 852
rend, 79, 299, 302, 308, 316, 320, 324, 331, 464, 809
renew, 178, 179
rep, 612, 614
replace, 80, 576
replace_copy, 577
replace_copy_if, 577
replace_if, 578
repositioning, 102, 109
reserve, 80, 302, 341
reserved identifiers, 56
reset, 477, 481
resetiosflags, 95
resize, 80, 302, 308, 316
resource: stealing, 205
responsibility of the programmer, 299, 301, 305, 311, 315, 335, 477
restrictions, 10
result_of, 765
result_type, 606, 842
rethrow_exception, 673
return, 217
return by argument, 36
return_type: implicit, 488
return_type: late-specified, 488
return_type: via result_of, 765
return_value, 14, 285
return_value_optimization, 213
reusable software, 388, 407
reverse, 308, 579
reverse_iterator, 809
Reverse Polish Notation, 334
reverse_copy, 580
reverse_iterator, 299, 302, 307, 316, 320, 324, 330, 809
reverse_iterator: initialized by iterator, 809
reversed_iterator, 463
rfind, 80
right, 96
right-hand, 294, 296
rmExceptFd, 908
rmReadFd, 908
rmWriteFd, 908
RNG, 508
rotate, 580
rotate_copy, 581
round_error, 712
round_style, 712
RPN, 334
rule of thumb, 13, 20, 27, 55, 63, 137, 152, 155, 160, 179, 244, 245, 263, 267, 277, 304, 357, 358, 387, 433, 486, 487, 493, 683, 703, 714, 735, 749, 853
run-time, 381, 399, 708
run-time error, 233
run-time support system, 189
run-time vs. compile-time, 823
runtime_error, 237
rvalue, 39, 257, 267, 465, 476
rvalue reference, 39
rvalue reference return type, 204
rvalue: distinguish from lvalue, 927
sbumpc, 407, 884
scalar type, 346
scanf, 108
scanner, 877, 933, 935, 938
scanner generator, 933
ScannerBase, 935
scientific, 98
scientific notation, 98
scope resolution operator, 25, 59, 276, 368, 372, 393
scope: class, 432
scope: global, 432
search, 582
search_n, 583
second, 296
second_argument_type, 606, 842
seconds, 612
seek beyond file boundaries, 102, 110
seek_dir, 448
seek_off, 886
seekdir, 102, 110, 409
seekg, 109
seekoff, 414, 885
seekp, 102
seekpos, 414, 885, 886
segmentation fault, 474
select, 905
Selector, 906
set, 328
set: header file, 328, 331
set_difference, 584
set_intersection, 585
set_new_handler, 188
set_symmetric_difference, 587
INDEX

set_union, 588
setAlarm, 907
setbase, 97
setbuf, 413
setDebug, 935
setf, 95
setfill, 94
setg, 411, 881, 882, 884, 886
setiosflags, 95
setjmp, 217, 220
setp, 413, 879
setprecision, 95
setstate, 93
setup.exe, 9
setw, 95
SFINAE, 719
sgetc, 408
sgetn, 408, 885
shadow member, 362
shared state, 653
shared state: abandon, 654
shared state: destruction, 654
shared state: releasing, 654
shared_future, 658
shared_ptr, 55, 478, 804
shared_ptr: 0-pointer, 479
shared_ptr: default, 479
shared_ptr: defining, 478
shared_ptr: initialization, 479
shared_ptr: operators, 480
shared_ptr: used type, 480
showbase, 97
showmanyc, 411
showpoint, 98
showpos, 97
shrink_to_fit, 81, 302, 317
shuffling, 526
signal, 899
signaling_NaN, 712
sin, 347
single inheritance, 370
Singleton, 239
sinh, 347
size, 81, 144, 299, 303, 308, 311, 314, 317, 324, 331, 335, 341
size specification, 165
size: of pointers to members, 438
size_t, 51, 274
size_type, 70
sizeof, 8, 161, 175, 181, 753, 846
sizeof derived vs base classes, 376
skipping leading blanks, 30
sleep_for, 618
sleep_until, 618
slicing, 374
snexce, 408
socket, 88, 877
sort, 308, 458, 589
sort criteria: hierarchic sorting, 783
sort: multiple hierarchal criteria, 593
sort_heap, 603
sorted collection of value, 331
sorted collection of values, 328
sorting, 526
splice, 308
split buffer, 412
sprintf, 100
sputback, 408
sputc, 408
sputn, 408
sqrt, 347
sscanf, 107
sstream: header file, 89, 105, 111
stable_partition, 590
stable_sort, 591, 783
stack, 333
stack constructors, 334
stack data structure, 333
stack operations, 284
stack: header file, 333
standard exceptions, 236
standard layout, 215
standard namespace, 25
standard namespace: and STL, 453
standard normal distribution, 518
Standard Template Library, 453
standard-layout, 215
stat, 50, 140
state flags, 235
state of I/O streams, 86, 89
static, 11, 58, 163
static binding, 381, 383
static data members: initialization, 164
static data: const, 166
static inline member functions, 171
static member functions, 170
static members, 163, 737
static object, 126
static polymorphism, 721, 790, 839
static polymorphism: downsides, 793
static type checking, 398
static type identification, 399
static variable: initialization, 631
static: data members, 163
static: members, 437
static_assert, 708
static_cast, 52, 375, 697
static_pointer_cast, 482
std, 85
std namespace: off limits, 593
std::distance, 463
std::move, 207
std::streambuf, 881
std::system_error, 237
std::u16string, 70
std::u32string, 70
std::wstring, 70
std=c++14, 9
stderr, 29
STDERR_FILENO, 902
stdexcept, 80
stdexcept: header file, 236
stdin, 29
STDIN_FILENO, 902
stdio.h, 14
stdout, 29
STDOUT_FILENO, 880
steady_clock, 614
STL, 453
STL: required types, 919
stod, 82
stof, 81
stoi, 82
stold, 82
string, as union member, 941
string: declaring, 70
string: header file, 70, 81
string: iterator types, 73
string::iterator, 441
string::size_type, 70
strong guarantee, 243
Stroustrup, 27
struct, 22
struct: empty, 856
Structured Computer Organization, 644
Stuart, A. & Ord, K, 507
student_t_distribution<RealType = double>, 520
substitution failure, 719
substr, 81
subtract_with_carry_engine, 508
sungetc, 408
Sutter, H., 244
swap, 81, 198, 246, 299, 303, 309, 317, 324, 331, 341, 477, 481, 593, 623, 635
swap area, 188
swap_ranges, 594
swapping, 526
Swiss army knife, 370
symbol area, 935
symbolic constants, 30
sync, 414, 878, 880
syntactic elements, 218
system, 895, 899
system_category, 241
system_clock, 614
system_clock::from_time_t, 614
system_clock::to_time_t, 614
system_error: header file, 237–239
tag, 348
TagTrait, 950
Tanenbaum, A.S., 644
task switching, 609
TCP/IP stack, 407
tellg, 109
tellp, 102
template, 85, 453, 681, 723
template alias, 840
template declaration, 695
template explicit specialization, 704
template explicit type specification: omitting, 707
template header, 681
template header: for member templates, 727
template instantiation declaration, 707
template mechanism, 679, 680
template members: without template type parameters, 818
typename, 816
typename ...Params, 752
typename & & , 39
typename vs. class, 836
typename: and template subtypes, 739
typename: disambiguating code, 738
types: required by STL, 919
U , 49
uflow, 408, 411
uint32_t , 52
Ullman, J .D., 304
unary function object, 460
unary not, 919
unary operator , 919
unary predicate, 533
unary_function, 919
unbound friend, 770, 776
undefined reference to vtable, 421
underflow, 411, 881
underflow_error, 237
unget, 109, 887
Unicode, 51
uniform initialization, 142
uniform_int_distribution< IntType = int>, 520
uniform_real_distribution< RealType = double>, 521
unimplemented: mangling dotstar_expr, 686
union, 22, 941
union: polymorphic, 952
unique, 309, 481, 597
unique_copy, 598
unique_lock, 633, 634
unique_ptr, 471, 804
unique_ptr: 0-pointer, 473
unique_ptr: assignment, 474
unique_ptr: default, 473
unique_ptr: defining, 472
unique_ptr: initialization, 474
unique_ptr: move constructor, 473
unique_ptr: operators, 476
unique_ptr: reaching members, 475
unique_ptr: used type, 475
unistd.h: header file, 878, 881, 882, 885, 901
unistdbuf, 99
Unix, 114, 117, 899, 900, 938, 947
unlock, 630, 636
unnamed type, 688
unordered_map, 335, 336
unordered_map: header file, 335
unordered_map: member functions, 339
unordered_map::hasher, 336
unordered_map::key_equal, 336
unordered_map::key_type, 336
unordered_map::mapped_type, 336
unordered_multimap, 342
unordered_multiset, 344
unordered_set, 343
unordered_set: header file, 343
unordered_set::hasher, 343
unordered_set::key_equal, 343
unordered_set::key_type, 343
unpack operator, 753, 761
unrestricted union, 347
unsetf, 95
unsigned int, 51
upper_bound, 324, 331, 599
uppercase, 98
URNG, 509
use_count, 481
user interface, 383
user-defined literal, 170
user-defined literals, 289
using, 46, 160
using declaration, 325
using directive, 60
using namespace std, 25
using vs. templates, 684
using: in derived classes, 368
using: restrictions, 64
UTF-16, 51
UTF-32, 51
UTF-8, 51
utility, 207
utility: header file, 296, 756
vague linkage, 153
valid state, 70
value, 317
value_comp, 325, 331
value_type, 317, 328
ValueType, 336
Vandevoorde, D., 815
variadic functions, 752
variadic non-type parameters, 762
variadic template: number of arguments, 753
variadic templates, 752
vector, 300, 463
vector constructors, 300
vector: header file, 300
vector: member functions, 300
viable functions, 715
virtual, 383
virtual base class, 393
virtual constructor, 422
virtual derivation, 394
virtual destructor, 387, 388
virtual member function, 383
virtual: vs static, 163
visibility: nested classes, 441
void, 17
void *, 229, 274, 276, 278
volatile, 691
vpointer, 418
vprintf, 101
vscanf, 108
vtable, 418, 839
vtable: and multiple inheritance, 420
vtable: undefined reference, 421

wait, 641, 643, 906
wait_for, 641, 643
wait_until, 641, 643
waitpid, 897
wchar_t, 48, 50, 85
weibull_distribution<RealType = double>, 522
what, 235, 236
white space, 30, 99, 100
width, 95
wild pointer, 190, 226
Williams, A., 609
wrapper, 187, 593, 891
wrapper class, 88, 268, 373
wrapper functions, 172
wregex, 495
write, 101
write beyond end of file, 102
writeFd, 907
ws, 100

X-windows, 51
xor, 290
xor_eq, 290
XQueryPointer, 51
xsgetn, 408, 412, 882, 884, 885
xsputn, 408, 413

yacc, 933
yield, 618
yocto, 611
Yodl, 3
yotta, 611

zepto, 611
zero, 613
zetta, 611
zombie, 899, 911