

# C++ Annotations Version 5.1.1a

Frank B. Brokken  
Computing Center, University of Groningen  
Landleven 1,  
P.O. Box 800,  
9700 AV Groningen  
The Netherlands  
Published at the University of Groningen  
ISBN 90 367 0470 7

1994 - 2001

### **Abstract**

This document is intended for knowledgeable users of **C** who would like to make the transition to **C++**. It is a guide for Frank's **C++** programming courses, which are given yearly at the University of Groningen. As such, this document is not a complete **C++** handbook. Rather, it serves as an addition to other documentation sources.

If you want a **hard-copy version of the C++ annotations**: hard copies are available in postscript, pdf and other formats in

`ftp://ftp.rug.nl/contrib/frank/documents/cplusplus.annotations,`

in files having names starting with `cplusplus`.

# Contents

<b>1</b>	<b>Overview of the chapters</b>	<b>13</b>
<b>2</b>	<b>Introduction</b>	<b>14</b>
2.1	What's new in the C++ Annotations . . . . .	14
2.2	The history of C++ . . . . .	16
2.2.1	History of the C++ Annotations . . . . .	16
2.2.2	Compiling a C program by a C++ compiler . . . . .	17
2.2.3	Compiling a C++ program . . . . .	18
2.3	Advantages and pretensions of C++ . . . . .	19
2.4	What is Object-Oriented Programming? . . . . .	20
2.5	Differences between C and C++ . . . . .	21
2.5.1	Namespaces . . . . .	21
2.5.2	End-of-line comment . . . . .	22
2.5.3	NULL-pointers vs. 0-pointers . . . . .	22
2.5.4	Strict type checking . . . . .	22
2.5.5	A new syntax for casts . . . . .	23
2.5.6	The 'void' parameter list . . . . .	25
2.5.7	The '#define __cplusplus' . . . . .	25
2.5.8	The usage of standard C functions . . . . .	25
2.5.9	Header files for both C and C++ . . . . .	26
2.5.10	The definition of local variables . . . . .	27
2.5.11	Function Overloading . . . . .	28
2.5.12	Default function arguments . . . . .	29
2.5.13	The keyword 'typedef' . . . . .	30
2.5.14	Functions as part of a struct . . . . .	31

<b>3</b>	<b>A first impression of C++</b>	<b>32</b>
3.1	More extensions to C in C++	32
3.1.1	The scope resolution operator ::	32
3.1.2	'cout', 'cin' and 'cerr'	33
3.1.3	The keyword 'const'	34
3.1.4	References	36
3.2	Functions as part of structs	40
3.3	Several new data types	41
3.3.1	The 'bool' data type	41
3.3.2	The 'wchar_t' data type	42
3.4	Data hiding: public, private and class	43
3.5	Structs in C vs. structs in C++	45
3.6	Namespaces	46
3.6.1	Defining namespaces	46
3.6.2	Referring to entities	47
3.6.3	The standard namespace	51
3.6.4	Nesting namespaces and namespace aliasing	52
<b>4</b>	<b>The 'string' data type</b>	<b>57</b>
4.1	Operations on strings	57
4.2	Overview of operations on strings	66
4.2.1	The string initializers	67
4.2.2	The string iterators	67
4.2.3	The string operators	67
4.2.4	The string member functions	68
<b>5</b>	<b>The IO-stream Library</b>	<b>76</b>
5.1	Iostream header files	79
5.2	The foundation: the class 'ios_base'	80
5.3	Interfacing 'streambuf' objects: the class 'ios'	80
5.3.1	Condition states	81
5.3.2	Formatting output and input	83
5.4	Output	87

5.4.1	Basic output: the class 'ostream' . . . . .	87
5.4.2	Output to files: the class 'ofstream' . . . . .	89
5.4.3	Output to memory: the class 'ostringstream' . . . . .	91
5.5	Input . . . . .	93
5.5.1	Basic input: the class 'istream' . . . . .	93
5.5.2	Input from streams: the class 'ifstream' . . . . .	96
5.5.3	Input from memory: the class 'istringstream' . . . . .	97
5.6	Manipulators . . . . .	98
5.7	The 'streambuf' class . . . . .	101
5.7.1	Protected 'streambuf' members . . . . .	103
5.7.2	The class 'filebuf' . . . . .	106
5.8	Advanced topics . . . . .	107
5.8.1	Copying streams . . . . .	107
5.8.2	Coupling streams . . . . .	108
5.8.3	Redirection using streams . . . . .	109
5.8.4	Reading AND Writing to a stream . . . . .	111
<b>6</b>	<b>Classes</b>	<b>119</b>
6.1	The constructor . . . . .	120
6.1.1	A first application . . . . .	122
6.1.2	Constructors: with and without arguments . . . . .	124
6.2	Const member functions and const objects . . . . .	127
6.3	The keyword 'inline' . . . . .	128
6.3.1	Inline functions within class declarations . . . . .	129
6.3.2	Inline functions outside of class declarations . . . . .	129
6.3.3	When to use inline functions . . . . .	130
6.4	Objects in objects: composition . . . . .	131
6.4.1	Composition and const objects: const member initializers . . . . .	131
6.4.2	Composition and reference objects: reference member initializers . . . . .	132
6.5	Header file organization with classes . . . . .	134
6.5.1	Using namespaces in header files . . . . .	138
<b>7</b>	<b>Classes and memory allocation</b>	<b>140</b>

7.1	The operators 'new' and 'delete' . . . . .	141
7.1.1	Allocating arrays . . . . .	142
7.1.2	Deleting arrays . . . . .	142
7.1.3	Enlarging arrays . . . . .	143
7.2	The destructor . . . . .	144
7.2.1	New and delete and object pointers . . . . .	146
7.2.2	The function set_new_handler() . . . . .	150
7.3	The assignment operator . . . . .	151
7.3.1	Overloading the assignment operator . . . . .	153
7.4	The this pointer . . . . .	156
7.4.1	Preventing self-destruction with this . . . . .	156
7.4.2	Associativity of operators and this . . . . .	157
7.5	The copy constructor: Initialization vs. Assignment . . . . .	158
7.5.1	Similarities between the copy constructor and operator=() . . . . .	162
7.5.2	Preventing the use of certain member functions . . . . .	163
7.6	Conclusion . . . . .	164
<b>8</b>	<b>Exceptions</b>	<b>165</b>
8.1	Using exceptions: syntax elements . . . . .	166
8.2	An example using exceptions . . . . .	166
8.2.1	No exceptions: 'setjmp()' and 'longjmp()' . . . . .	168
8.2.2	Exceptions: the preferred alternative . . . . .	170
8.3	Throwing exceptions . . . . .	172
8.3.1	The empty 'throw' statement . . . . .	175
8.4	The try block . . . . .	177
8.5	Catching exceptions . . . . .	177
8.5.1	The default catcher . . . . .	179
8.6	Declaring exception throwers . . . . .	180
8.7	Iostreams and exceptions . . . . .	182
<b>9</b>	<b>More Operator Overloading</b>	<b>184</b>
9.1	Overloading 'operator[]()' . . . . .	184
9.2	overloading the insertion and extraction operators . . . . .	187

9.3	Conversion operators	188
9.4	The 'explicit' keyword	191
9.5	Overloading increment and decrement	193
9.6	Overloading 'operator new(size_t)'	196
9.7	Overloading 'operator delete(void *)'	198
9.8	Operators 'new[]' and 'delete[]'	199
9.9	Function Objects	202
9.9.1	Constructing manipulators	205
9.10	Overloadable Operators	207
<b>10</b>	<b>Static data and functions</b>	<b>208</b>
10.1	Static data	208
10.1.1	Private static data	209
10.1.2	Public static data	211
10.2	Static member functions	211
<b>11</b>	<b>Friends</b>	<b>213</b>
11.1	Friend-functions	214
11.2	Inline friends	215
<b>12</b>	<b>Abstract Containers</b>	<b>218</b>
12.1	The 'pair' container	220
12.2	Sequential Containers	221
12.2.1	The 'vector' container	221
12.2.2	The 'list' container	224
12.2.3	The 'queue' container	231
12.2.4	The 'priority_queue' container	232
12.2.5	The 'deque' container	234
12.2.6	The 'map' container	237
12.2.7	The 'multimap' container	244
12.2.8	The 'set' container	246
12.2.9	The 'multiset' container	249
12.2.10	The 'stack' container	251

12.2.11 The ‘hash_map’ and other hashing-based containers . . . . .	253
12.3 The ‘complex’ container . . . . .	257
<b>13 Inheritance</b>	<b>260</b>
13.1 Related types . . . . .	261
13.2 The constructor of a derived class . . . . .	264
13.3 The destructor of a derived class . . . . .	264
13.4 Redefining member functions . . . . .	265
13.5 Multiple inheritance . . . . .	267
13.6 Conversions between base classes and derived classes . . . . .	270
13.6.1 Conversions in object assignments . . . . .	270
13.6.2 Conversions in pointer assignments . . . . .	271
<b>14 Polymorphism</b>	<b>273</b>
14.1 Virtual functions . . . . .	273
14.2 Virtual destructors . . . . .	275
14.3 Pure virtual functions . . . . .	276
14.4 Virtual functions in multiple inheritance . . . . .	277
14.4.1 Ambiguity in multiple inheritance . . . . .	278
14.4.2 Virtual base classes . . . . .	279
14.4.3 When virtual derivation is not appropriate . . . . .	281
14.5 Run-Time Type identification . . . . .	283
14.5.1 The dynamic_cast operator . . . . .	283
14.5.2 The typeid operator . . . . .	285
14.6 Deriving classes from ‘streambuf’ . . . . .	287
14.7 A polymorphic exception class . . . . .	292
14.8 How polymorphism is implemented . . . . .	294
<b>15 Classes having pointers to members</b>	<b>296</b>
15.1 Pointers to members: an example . . . . .	296
15.2 Defining pointers to members . . . . .	297
15.3 Using pointers to members . . . . .	299
15.4 Pointers to static members . . . . .	302



<b>16 Nested Classes</b>	<b>303</b>
16.1 Defining nested class members . . . . .	305
16.2 Declaring nested classes . . . . .	306
16.3 Accessing private members in nested classes . . . . .	306
16.4 Nesting enumerations . . . . .	310
16.4.1 Empty enumerations . . . . .	312
<b>17 The Standard Template Library, generic algorithms</b>	<b>313</b>
17.1 Predefined function objects . . . . .	313
17.1.1 Arithmetic Function Objects . . . . .	315
17.1.2 Relational Function Objects . . . . .	319
17.1.3 Logical Function Objects . . . . .	320
17.1.4 Function Adaptors . . . . .	320
17.2 Iterators . . . . .	322
17.2.1 Insert iterators . . . . .	325
17.2.2 istream iterators . . . . .	325
17.2.3 ostream iterators . . . . .	327
17.3 The 'auto_ptr' class . . . . .	329
17.3.1 Defining auto_ptr variables . . . . .	330
17.3.2 Pointing to a newly allocated object . . . . .	330
17.3.3 Pointing to another auto_ptr . . . . .	331
17.3.4 Creating a plain auto_ptr . . . . .	332
17.3.5 Auto_ptr: operators and members . . . . .	333
17.4 The Generic Algorithms . . . . .	334
17.4.1 accumulate() . . . . .	335
17.4.2 adjacent_difference() . . . . .	336
17.4.3 adjacent_find() . . . . .	337
17.4.4 binary_search() . . . . .	338
17.4.5 copy() . . . . .	339
17.4.6 copy_backward() . . . . .	340
17.4.7 count() . . . . .	341
17.4.8 count_if() . . . . .	342

17.4.9	<code>equal()</code>	343
17.4.10	<code>equal_range()</code>	344
17.4.11	<code>fill()</code>	346
17.4.12	<code>fill_n()</code>	346
17.4.13	<code>find()</code>	347
17.4.14	<code>find_if()</code>	348
17.4.15	<code>find_end()</code>	349
17.4.16	<code>find_first_of()</code>	351
17.4.17	<code>for_each()</code>	352
17.4.18	<code>generate()</code>	354
17.4.19	<code>generate_n()</code>	355
17.4.20	<code>includes()</code>	355
17.4.21	<code>inner_product()</code>	357
17.4.22	<code>inplace_merge()</code>	359
17.4.23	<code>iter_swap()</code>	360
17.4.24	<code>lexicographical_compare()</code>	361
17.4.25	<code>lower_bound()</code>	363
17.4.26	<code>max()</code>	364
17.4.27	<code>max_element()</code>	365
17.4.28	<code>merge()</code>	366
17.4.29	<code>min()</code>	368
17.4.30	<code>min_element()</code>	369
17.4.31	<code>mismatch()</code>	369
17.4.32	<code>next_permutation()</code>	371
17.4.33	<code>nth_element()</code>	373
17.4.34	<code>partial_sort()</code>	374
17.4.35	<code>partial_sort_copy()</code>	375
17.4.36	<code>partial_sum()</code>	376
17.4.37	<code>partition()</code>	377
17.4.38	<code>prev_permutation()</code>	378
17.4.39	<code>random_shuffle()</code>	379
17.4.40	<code>remove()</code>	381

17.4.41	<code>remove_copy()</code>	382
17.4.42	<code>remove_if()</code>	383
17.4.43	<code>remove_copy_if()</code>	384
17.4.44	<code>replace()</code>	384
17.4.45	<code>replace_copy()</code>	385
17.4.46	<code>replace_if()</code>	386
17.4.47	<code>replace_copy_if()</code>	387
17.4.48	<code>reverse()</code>	388
17.4.49	<code>reverse_copy()</code>	388
17.4.50	<code>rotate()</code>	389
17.4.51	<code>rotate_copy()</code>	390
17.4.52	<code>search()</code>	391
17.4.53	<code>search_n()</code>	392
17.4.54	<code>set_difference()</code>	393
17.4.55	<code>set_intersection()</code>	394
17.4.56	<code>set_symmetric_difference()</code>	396
17.4.57	<code>set_union()</code>	397
17.4.58	<code>sort()</code>	398
17.4.59	<code>stable_partition()</code>	399
17.4.60	<code>stable_sort()</code>	400
17.4.61	<code>swap()</code>	402
17.4.62	<code>swap_ranges()</code>	403
17.4.63	<code>transform()</code>	404
17.4.64	<code>unique()</code>	406
17.4.65	<code>unique_copy()</code>	407
17.4.66	<code>upper_bound()</code>	408
17.4.67	Heap algorithms	410
<b>18</b>	<b>Templates</b>	<b>414</b>
18.1	Template functions	414
18.1.1	Template function definitions	415
18.1.2	Instantiations of template functions	417

18.1.3	Argument deduction . . . . .	420
18.1.4	Explicit arguments . . . . .	424
18.1.5	Template explicit specialization . . . . .	425
18.1.6	Overloading template functions . . . . .	426
18.1.7	Selecting an overloaded (template) function . . . . .	428
18.1.8	Name resolution within template functions . . . . .	429
18.2	Template classes . . . . .	430
18.2.1	Template class definitions . . . . .	431
18.2.2	Template class instantiations . . . . .	433
18.2.3	Non-type parameters . . . . .	434
18.2.4	Template class member functions . . . . .	434
18.2.5	Template classes and friend declarations . . . . .	435
18.2.6	Template classes and static data . . . . .	438
18.2.7	Derived Template Classes . . . . .	439
18.2.8	Nesting and template classes . . . . .	440
18.2.9	Member templates . . . . .	442
18.2.10	Template class specializations . . . . .	444
18.2.11	Template class partial specializations . . . . .	446
18.2.12	Name resolution within template classes . . . . .	448
18.3	Constructing iterators . . . . .	449
<b>19</b>	<b>Concrete examples of C++</b>	<b>451</b>
19.1	'streambuf' classes using file descriptors . . . . .	451
19.1.1	A class for output operations . . . . .	451
19.1.2	Classes for input operations . . . . .	455
19.2	Using form() with ostream objects . . . . .	465
19.3	Redirection revisited . . . . .	466
19.4	The fork() system call . . . . .	467
19.4.1	The 'ChildProcess' abstract base class . . . . .	468
19.4.2	The 'ParentProcess' abstract base class . . . . .	469
19.4.3	The 'Redirector' abstract base class . . . . .	470
19.4.4	The 'Fork' class: implementation . . . . .	471

19.4.5	Support classes . . . . .	472
19.4.6	The 'Pipe' classes . . . . .	473
19.4.7	The 'IORedirector' . . . . .	475
19.4.8	The 'ChildPlain' class . . . . .	477
19.4.9	The 'ParentCmd' class . . . . .	477
19.4.10	The 'ParentSlurp' class . . . . .	479
19.4.11	The 'Fork' class: example of use . . . . .	480
19.4.12	The 'Daemon' program . . . . .	481
19.5	Function objects performing bitwise operations . . . . .	482
19.6	Implementing a reverse_iterator . . . . .	483
19.7	Using Bison and Flex . . . . .	488
19.7.1	Using Flex++ to create a scanner . . . . .	489
19.7.2	Using both bison++ and flex++ . . . . .	497

# 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: The 'string' data type.
- Chapter 5: The C++ I/O library.
- Chapter 6: The 'class' concept: structs having functions. The 'object' concept: variables of a class.
- Chapter 7: Allocation and returning unused memory: `new`, `delete`, and the function `set_new_handler()`.
- Chapter 8: Exceptions: handle errors where appropriate, rather than where they occur.
- Chapter 9: Give your own meaning to operators.
- Chapter 10: Static data and functions: members of a class not bound to objects.
- Chapter 11: Gaining access to private parts: friend functions and classes.
- Chapter 12: Abstract Containers to put stuff into.
- Chapter 13: Building classes upon classes: setting up class hierarcies.
- Chapter 14: Changing the behavior of member functions accessed through base class pointers.
- Chapter 15: Classes having pointers to members: pointing to locations inside objects.
- Chapter 16: Constructing classes and enums within classes.
- Chapter 17: The Standard Template Library, generic algorithms.
- Chapter 18: Templates: using *molds* for code that is type dependent.
- Chapter 19: Several examples of programs written in C++.

## Chapter 2

# Introduction

This document presents an introduction to programming in C++. It is a guide for C/C++ programming courses, that Frank gives yearly at the University of Groningen. As such, this document is not a complete C/C++ handbook, but rather serves as an addition to other documentation sources<sup>1</sup>.

The reader should realize that extensive knowledge of the C programming language is assumed and required. This document continues where topics of the C programming language end, such as pointers, memory allocation and compound types.

The version number of this document (currently 5.1.1a) is updated when the contents of the document change. The first number is the major number, and will probably not be changed 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 Computing Center, University of Groningen, the Netherlands. This document was typeset using the yodl formatting system.

**All rights reserved. No part of this document may be published or changed without prior consent of the author. Direct all correspondence concerning suggestions, additions, improvements or changes to this document to the author:**

**Frank B. Brokken  
Computing Center, University of Groningen  
Landleven 1,  
P.O. Box 800,  
9700 AV Groningen  
The Netherlands  
(email: f.b.brokken@rc.rug.nl)**

In this chapter a first impression of C++ is presented. A few extensions to C are reviewed and a tip of the mysterious veil surrounding object oriented programming (OOP) is lifted.

### 2.1 What's new in the C++ Annotations

This section is modified when the first or second part of the version number changes.

---

<sup>1</sup>e.g., the Dutch book *De programmeertaal C*, Brokken and Kubat, University of Groningen, 1996

- Version 5.1.1 was released after modifying the sections related to the `fork()` system call in chapter 19. Under the ANSI/ISO standard many of the previously available extensions (like `procbuf`, and `vform()`) applied to streams were discontinued. Starting with version 5.1.1, ways of constructing these facilities under the ANSI/ISO standard are discussed in the C++ Annotations. I consider the involved subject sufficiently complex to warrant the upgrade to a new subversion.
- With the advent of the Gnu g++ compiler version 3.00, a more strict implementation of the ANSI/ISO C++ standard became available. This resulted in version 5.1.0 of the Annotations, appearing shortly after version 5.0.0. In version 5.1.0 chapter 5 was modified and several cosmetic changes took place (e.g., removing `class` from template type parameter lists, see chapter 18). Intermediate versions (like 5.0.0a, 5.0.0b) were not further documented, but were mere intermediate releases while approaching version 5.1.0. Code examples will gradually be adapted to the new release of the compiler.

**In the meantime the reader should be prepared to insert**

```
using namespace std;
```

**in many code examples, just beyond the `#include` preprocessor directives as a temporary measure to make the example accepted by the compiler.**

- New insights develop all the time, resulting in version 5.0.0 of the Annotations. In this version a lot of old code was cleaned up and typos were repaired. According to current standard, *namespaces* are required in C++ programs, so they are introduced now very early (in section 2.5.1) in the Annotations. A new section about using external programs was added to the Annotations (and removed again in version 5.1.0), and the new `stringstream` class, replacing the `strstream` class is now covered too (sections 5.4.3 and 5.5.3). Actually, the chapter on input and output was completely rewritten. Furthermore, the operators `new` and `delete` are now discussed in chapter 7, where they fit better than in a chapter on classes, where they previously were discussed. Chapters were moved, split or reordered, to subjects could generally be introduced without forward references. Finally, the `html`, `PostScript` and `pdf` versions of the C++ Annotations now contain an index (sigh of relief ?) All in, considering the volume and nature of the modifications, it seemed right to upgrade to a full major version. So here it is.

Considering the volume of the annotations, I'm sure there will be typos found every now and then. Please do not hesitate to send me mail containing any mistakes you find or corrections you would like to suggest.

- In release 4.4.1b the `pagesize` in the LaTeX file was defined to be `din A4`. In countries where other pagesizes are standard the conversion the default `pagesize` might be a better choice. In that case, remove the `dina4` option from `cplusplus.tex` (or `cplusplus.yo` if you have `yodl` installed), and reconstruct the annotations from the `TeX`-file or `Yodl`-files.

The Annotations mailing lists was stopped at release 4.4.1d. From this point on only minor modifications were expected, which are not anymore generally announced.

At some point, I considered version 4.4.1 to be the final version of the C++ annotations. However, a section on special I/O functions was added to cover unformatted I/O, and the section about the `string` datatype had its layout improved and was, due to its volume, given a chapter of its own (chapter 4). All this eventually resulted in version 4.4.2.

Version 4.4.1 again contains new material, and reflects the ANSI/ISO standard (well, I try to have it reflect the ANSI/ISO standard). In version 4.4.1, several new sections and chapters were added, among which a chapter about the *Standard Template Library* (STL) and *generic algorithms*.

Version 4.4.0 (and subletters) was a mere construction version and was never made available.



The version 4.3.1a is a precursor of 4.3.2. In 4.3.1a most of the typos I've received since the last update have been processed. In version 4.3.2. extra attention was paid to the syntax for function address function addresses and pointers to member functions.

The decision to upgrade from version 4.2.\* to 4.3.\* was made after realizing that the lexical scanner function `yylex()` can be defined in the scanner class that is derived from `yyFlexLexer`. Under this approach the `yylex()` function can access the members of the class derived from `yyFlexLexer` as well as the public and protected members of `yyFlexLexer`. The result of all this is a clean implementation of the rules defined in the `flex++` specification file.

The upgrade from version 4.1.\* to 4.2.\* was the result of the inclusion of section 3.3.1 about the **bool** data type in chapter 3. The distinction between differences between C and C++ and extensions of the C programming languages is (albeit a bit fuzzy) reflected in the introduction chapter and the chapter on first impressions of C++: The introduction chapter covers some differences between C and C++, whereas the chapter about first impressions of C++ covers some extensions of the C programming language as found in C++.

Major version 4 represents a major rewrite of the previous version 3.4.14: The document was rewritten from SGML to Yodl and many new sections were added. All sections got a tune-up. The distribution basis, however, hasn't changed: see the introduction.

Modifications in versions 1.\*, 2.\*, and 3.\* were not logged.

Subreleases like 4.4.2a etc. contain bugfixes and typographical corrections.

## 2.2 The history of C++

The first implementation of C++ was developed in the nineteen-eighties 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, which converted special constructions in its source code to plain C. This code was then compiled by a normal 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 compiled and linked.

The nomenclature of C++ source files remains: the extensions `.cc` and `.cpp` are usually still used. However, the preliminary work of a C++ pre-compiler is in modern compilers usually included in the actual compilation process. Often compilers will determine the type of a source file by the 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 all possibilities of C, and more. This makes the transition from C to C++ quite easy. Programmers who are familiar with C may start 'programming in C++' by using source files with an extension `.cc` or `.cpp` instead of `.c`, and can then just comfortably slide into all the possibilities that C++ offers. No abrupt change of habits is required.

### 2.2.1 History of the C++ Annotations

The original version of the C++ Annotations was originally written by Frank 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.

In time several chapters were added, and the contents were modified thanks to countless readers who sent us their comment, due to which we were able to correct some typos and improve unclear parts.

The transition from major version three to major version four was realized by Frank: again new chapters were added, and the source-document was converted from SGML to Yodl(<http://www.xs4all.nl/~jantien/yodl/>)

The C++ Annotations are *not* freely distributable. Be sure to read the `legal` notes.

**Reading the annotations beyond this point implies that you are aware of the restrictions that we pose 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.

In the Internet, many useful hyperlinks exist to C++. Without even suggesting completeness (and without being checked regularly for existence: they might have died by the time you read this), the following might be worthwhile visiting:

- <http://www.cplusplus.com/ref/>: A reference site for C++.
- <http://www.cygnum.com/misc/wp/dec96pub/>: Makes available a version of the C++ ANSI/ISO standard.

## 2.2.2 Compiling a C program by a C++ compiler

For the sake of completeness, it must be mentioned here that C++ is 'almost' a superset of C. There are some small differences which you might encounter when you just rename a file to an extension `.cc` and run it through a C++ compiler:

- In C, `sizeof('c')` equals `sizeof(int)`, 'c' being any ASCII character. The underlying philosophy is probably that 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

```
putchar(10);
```

and

```
putchar('\n');
```

are synonyms.

In contrast, in C++, `sizeof('c')` is always 1 (but see also section 3.3.2), while an `int` is still an `int`. As we shall see later (see section 2.5.11), two function calls

```
somefunc(10);
```

and

```
somefunc('\n');
```

are quite separate functions: C++ discriminates functions by their arguments, which are different in these two calls: one function requires an `int` while the other one requires a `char`.

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

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

means in C that a function `func()` exists, which returns no value. However, in C, the declaration doesn't specify which arguments (if any) the function takes.

In contrast, such a declaration in C++ means that the function `func()` takes no arguments at all.

### 2.2.3 Compiling a C++ program

In order to compile a C++ program, a C++ compiler is needed. 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.linux.org>).

#### 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 going to the above URL for 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 normally what you would want.

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 the 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.

#### Compiling a C++ source text

In general, compiling a C++ source `source.cc` is done as follows:

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

This produces a binary program (`a.out` or `a.exe`). If the default name is not wanted, the name of the executable can be specified using the `-o` flag:

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

If only a compilation is required, the compiled module can be generated using the `-c` flag:

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

This produces the file `source.o`, which can be linked to other modules later on.

Using the `icmake` program a maintenance script can be used to assist in the construction and maintenance of C++ programs. This script has been tested on Linux platforms for several years now. Its description and components are found in a file named `icmake-C1.61.tar.gz` (or comparably), which is found in the same location as the `icmake` program. Alternatively, the standard `make` program can be used for maintenance of C++ programs. It is strongly advised to start using maintenance scripts or programs early in the study of the C++ programming language.

## 2.3 Advantages and pretensions of C++

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? In our opinion, C++ is a little overrated; in general this holds true for the entire object-oriented programming (OOP). The enthusiasm around C++ resembles somewhat the former allegations about Artificial-Intelligence (AI) languages like Lisp and Prolog: these languages were supposed to solve the most difficult AI-problems ‘almost without effort’. Obviously, too promising stories about any programming language must be overdone; in the end, each problem can be coded in any programming language (even BASIC or assembly language). The advantages or disadvantages of a given programming language aren’t in ‘what you can do with them’, but rather in ‘which tools the language offers to make the job easier’.

Concerning the above allegations of C++, we think that the following can be concluded. The development of new programs while existing code is reused can also be realized in C by, e.g., using function libraries: thus, handy functions can be collected in a library and need not be re-invented with each new program. Still, C++ offers its specific syntax possibilities for code reuse, apart from function libraries (see chapter 13).

Creating and using new data types is also very well 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.

Memory management is in principle in C++ as easy or as difficult as in C. Especially when dedicated C functions such as `xmalloc()` and `xrealloc()` are used<sup>2</sup>. In short, memory management in C or in C++ can be coded ‘elegantly’, ‘ugly’ or anything in between – this depends on the developer rather than on the language.

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

---

<sup>2</sup>these functions are often present in our C-programs, they allocate or abort the program when the memory pool is exhausted

disregard or heed a generated warning. 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 realized even in C; though it must be admitted that C++ offers special syntactical constructions. In contrast, programmers who prefer to use a global variable `int i` for each counter variable will quite likely not benefit from the concept of data hiding, be it in C or C++.

Concluding, C++ in particular and OOP in general are not solutions to all programming problems. C++, however, *does* offer some elegant syntactical possibilities which are worthwhile investigating. At the same time, the level of grammatical complexity of C++ has increased significantly compared to C. In time we got used to this increased level of complexity, but the transition didn’t take place fast or painless. With the Annotations we hope to help the reader to make the transition from C to C++ by providing, indeed, our *annotations* to what is found in some textbooks on C++. We hope you like this document and may benefit from it: Good luck!

## 2.4 What is Object-Oriented Programming?

Object-oriented programming propagates a slightly different approach to programming problems than the strategy which is usually used in C. The C-way is known as 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-oriented approach identifies the **key-words** in the problem. These keywords are then depicted in a diagram and arrows are drawn between these keywords to define an internal hierarchy. The keywords will be 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 some 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 as the same type, 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 extra functionality: the number of transactions (sales) and the bonus per transaction.

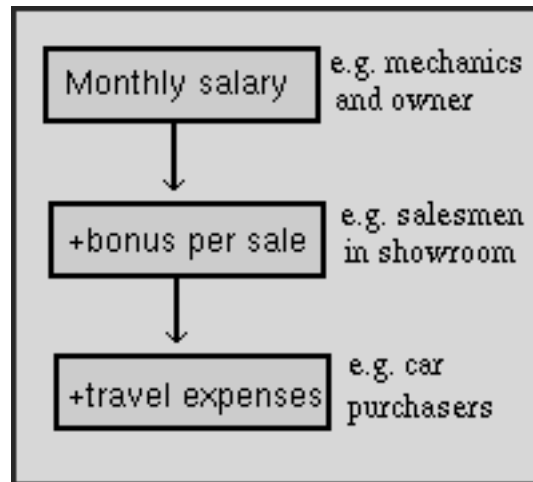


Figure 2.1: Hierarchy of objects in the salary administration.

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 the 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 thus identified objects 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. Subsequently more complex types are derived, while each derivation adds a little functionality. From these derived types, more complex types can be derived *ad infinitum*, until a representation of the entire problem can be made.

In C++ each of the objects can be represented in a *class*, containing the necessary functionality to do useful things with the variables (called *objects*) of these classes. Not all of the functionality and not all of the properties of a class is usually available to objects of other classes. As we will see, classes tend to *encapsulate* their properties in such a way that they are not immediately accessible from the outside world. Instead, dedicated functions are normally used to reach or modify the properties of objects.

## 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.

### 2.5.1 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* without losing the capability of using the standard `sin()` function, operating on *radians*.

Namespaces are covered extensively in section 3.6. 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 intend to actually compile the examples given in the Annotations, make sure that the sources start with the above using declaration.

### 2.5.2 End-of-line comment

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

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

    /*
       this is standard-C comment, over more
       than one line
    */
}
```

### 2.5.3 NULL-pointers vs. 0-pointers

In C++ all zero values are coded as 0. In C, where pointers are concerned, NULL is often used. This difference is purely stylistic, though one that is widely adopted. In C++ there’s no need anymore to use NULL. Indeed, according to the descriptions of the pointer-returning operator `new` 0 rather than NULL is returned when memory allocation fails.

### 2.5.4 Strict type checking

C++ uses very strict type checking. A prototype must be known for each function which is called, and the call must match the prototype. The program

```
int main()
{
    printf("Hello World\n");
}
```

does often compile under C, though with a warning that `printf()` is not a known function. Many C++ compilers will fail to produce code in such a situation<sup>3</sup>. The error is of course the missing `#include <stdio.h>` directive.

---

<sup>3</sup>When Gnu’s `g++` compiler encounters an unknown function, it assumes that an ‘ordinary’ C function is meant. It does complain however.

Although, while we're at it: in C++ the function `main()` *always* uses the `int` return value. It is possible to define `int main()`, without an explicit return statement, but a `return` statement without an expression cannot be given inside the `main()` function: a `return` statement in `main()` must always be given an `int`-expression. For example:

```
int main()
{
    return;      // won't compile: expects int expression
                // omitting the above statement is ok too
}
```

### 2.5.5 A new syntax for casts

Traditionally, C offers the following *cast* construction:

`(typename)expression`

in which `typename` is the name of a valid *type*, and `expression` an expression. Following that, C++ initially also supported the *function call style* cast notation:

`typename(expression)`

But, these casts are now called *old-style casts*, and they are deprecated. Instead, four *new-style casts* were introduced:

- The standard cast to convert one type to another is

`static_cast<type>(expression)`

- There is a special cast to do away with the `const` type-modification:

`const_cast<type>(expression)`

- A third cast is used to change the *interpretation* of information:

`reinterpret_cast<type>(expression)`

- And, finally, there is a cast form which is used in combination with polymorphism (see chapter 14): The

`dynamic_cast<type>(expression)`

is performed run-time to convert, e.g., a pointer to an object of a certain class to a pointer to an object in its so-called *class hierarchy*. At this point in the *Annotations* it is a bit premature to discuss the `dynamic_cast`, but we will return to this topic in section 14.5.1.



## The 'static\_cast'-operator

The `static_cast<type>(expression)` operator is used to convert one type to an acceptable other type. E.g., double to int. An example of such a cast is, assuming `intVar` is of type `int`:

```
intVar = static_cast<int>(12.45);
```

Another nice example of code in which it is a good idea to use the `static_cast<>()`-operator is in situations where the arithmetic assignment operators are used in mixed-type situations. E.g., consider the following expression (assume `doubleVar` is a variable of type `double`):

```
intVar += doubleVar;
```

Here, the evaluated expression actually is:

```
intVar = static_cast<int>(static_cast<double>(intVar) + doubleVar);
```

`intVar` is first promoted to a `double`, and is then added as `double` to `doubleVar`. Next, the sum is cast back to an `int`. These two conversions are a bit overdone. The same result is obtained by explicitly casting the `doubleVar` to an `int`, thus obtaining an `int`-value for the right-hand side of the expression:

```
intVar += static_cast<int>(doubleVar);
```

## The 'const\_cast'-operator

The `const_cast<type>(expression)` operator is used to do away with the `const`-ness of a (pointer) type. Assume that a function `fun(char *s)` is available, which performs some operation on its `char *s` parameter. Furthermore, assume that it's known that the function does not actually alter the string it receives as its argument. How can we use the function with a string like `char const hello[] = "Hello world"`?

Passing `hello` to `fun()` produces the warning

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

which can be prevented using the call

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

## The 'reinterpret\_cast'-operator

The `reinterpret_cast<type>(expression)` operator is used to reinterpret byte patterns. For example, the individual bytes making up a `double` value can easily be reached using `reinterpret_cast<>()`. Assume `doubleVar` is a variable of type `double`, then the individual bytes can be reached using

```
reinterpret_cast<char *>(&doubleVar)
```

This particular example also suggests the danger of the cast: it looks as though a standard C-string is produced, but there is not normally a trailing 0-byte. It's just a way to reach the individual bytes of the memory holding a double value.

More in general: using the cast-operators is a dangerous habit, as it suppresses the normal type-checking mechanism of the compiler. It is suggested to prevent casts if at all possible. If circumstances arise in which casts have to be used, document the reasons for their use well in your code, to make double sure that the cast is not the underlying cause for a program to misbehave.

### The 'dynamic\_cast'-operator

The `dynamic_cast<>()` operator is used in the context of polymorphism. The discussion of this cast is postponed until section 14.5.1.

## 2.5.6 The 'void' parameter list

A function prototype with an empty parameter list, such as

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

means in C that the argument list of the declared function is not prototyped: the compiler will not be able to warn against improper argument usage. When declaring a function in C which has no arguments, the keyword `void` is used, as in:

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

Because C++ enforces strict type checking, an empty parameter list is interpreted as the absence of any parameter. The keyword `void` can then be omitted: in C++ the above two declarations are equivalent.

## 2.5.7 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`.

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

## 2.5.8 The usage of 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()` which is a C function:

```
extern "C" void *xmalloc(unsigned 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:

```
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:

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

The above presented methods can be used without problem, but are not generally used. A more frequently used method to declare external C functions is presented next.

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

The combination of the predefined symbol `__cplusplus` and of 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:

```
#ifdef __cplusplus
extern "C"
{
#endif
    // declaration of C-data and functions are inserted here. E.g.,
    extern void *xmalloc(unsigned size);

#ifdef __cplusplus
}
#endif
```

Using this setup, a normal C header file is enclosed by `extern "C" {` which occurs at the start of the file and by `}`, which occurs at the end 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++.

An extra addition which is often seen is the following. Usually it is desirable to avoid multiple inclusions of the same header file. This can easily be achieved by including an `#ifndef` directive in the header file. An example of a file `myheader.h` would then be:

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

#endif
```

When this file is scanned for the first time 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 for a second time during the same compilation, the symbol `_MYHEADER_H_` is defined. All information between the `#ifndef` and `#endif` directives is skipped.

The symbol name `_MYHEADER_H_` serves in this context 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 preprocessor directive `#include <iostream>`, rather than `#include <iostream.h>` in a source. In the Annotations this convention is used with the standard C++ header files, but not everywhere else (Frankly, we tend not to follow this convention: our C++ header files still have the `.h` extension, and apparently nobody cares...).

There is more to be said about header files. In section 6.5 the preferred organization of header files with C++ classes is discussed.

## 2.5.10 The definition of 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 in some statements, just prior to their usage. A typical example is the `for` statement:

```
#include <stdio.h>

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

In this code fragment the variable `i` is created inside 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 compilers, the variable continues to exist after the execution of the `for`-statement, but a warning like

```
warning: name lookup of 'i' changed for new ANSI 'for' scoping using obsolete binding
at 'i'
```

will be 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's to be used after that statement, otherwise the variable can be defined at the `for`-statement itself.

Defining local variables when they're needed requires a little getting used to. However, eventually it tends to produce more readable 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 defined at the beginning of a function, following the first {,
- or they 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.

### 2.5.11 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. 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 *val)
{
    printf("String: %s\n", val);
}

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

In the above fragment three functions `show()` are defined, which only differ in their parameter lists: `int`, `double` and `char *`. The functions have the same names. The definition of several functions having identical names is called ‘function overloading’.

It is interesting that the way in which the C++ compiler implements function overloading is quite simple. Although the functions share the same name in the source text (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 name `void show (int)` to the internal name `vshowI`, while an analogous function with a `char*` argument might be called `vshowCP`. The actual names which are internally used 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.

A few remarks concerning function overloading are:

- The usage of more than one function with the same name but quite different actions should be avoided. 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 two functions have nothing in common except for their name. It would therefore be more practical to use names which suggest the action; say, `findname()` and `getvidmode()`.

- C++ does not allow that several functions only differ in their return value. This has the reason that it is always the programmer's choice to inspect or ignore the return value of a function. E.g., the fragment

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

holds no information concerning the return value of the function `printf()`<sup>4</sup>. Two functions `printf()` which would only differ in their return type could therefore not be distinguished by the compiler.

- Function overloading can produce surprises. E.g., imagine a statement like

```
show(0);
```

given the three functions `show()` above. The zero could be interpreted here as a `NULL` pointer to a `char`, i.e., a `(char *)0`, or as an integer with the value zero. C++ will choose to call the function expecting an integer argument, which might not be what one expects.

## 2.5.12 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:

```
#include <stdio.h>

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

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: the compiler will supply the missing argument when not specified. The code of the program becomes by no means shorter or more efficient.

Functions may be defined with more than one default argument:

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

---

<sup>4</sup>The return value is, by the way, an integer which states the number of printed characters. This return value is practically never inspected.

```

    ...
}

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 when necessary. A statement as `two_ints(,6)` is however not allowed: when arguments are omitted they must be on the right-hand side.

Default arguments must be known to the compiler when the code is generated where the arguments may have to be supplied. Often this means that the default arguments are present in a header file:

```

// sample header file
extern void two_ints(int a = 1, int b = 4);

// code of function in, say, two.cc
void two_ints(int a, int b)
{
    ...
}

```

Note that supplying the default arguments in function definitions instead of in the header file is not the correct approach: the compiler will read the header file and not the function definition when the function is used in other sources. Consequently, in that case no default arguments can be inserted by the compiler.

### 2.5.13 The keyword ‘typedef’

The keyword `typedef` is still allowed in C++, but no longer necessary when used as a prefix in union, struct or enum definitions. This is illustrated in the following example:

```

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):

```

somestruct
    what;

what.d = 3.1415;

```

### 2.5.14 Functions as part of a struct

In C++ it is allowed to define functions as part of a struct. This is the first concrete example of the definition of an object: as was described previously (see section 2.4), an object is a structure containing all involved code and data.

A definition of a struct `point` is given in the code fragment below. In this structure, two `int` data fields and one function `draw()` are declared.

```
struct point          // definition of a screen
{                    // dot:
    int
        x,          // coordinates
        y;          // x/y
    void
        draw(void); // drawing function
};
```

A similar structure could be part of a painting program and could, e.g., represent a pixel in the drawing. Concerning this struct it should be noted that:

- The function `draw()` which occurs in the struct definition is only a *declaration*. The actual code of the function, or in other words the actions which the function should perform, are located elsewhere: in the code section of the program, where all code is collected. We will describe the actual definitions of functions inside structs later (see section 3.2).
- The size of the struct `point` is just two ints. Even though a function is declared in the structure, its size is not affected by this. The compiler implements this behavior by allowing the function `draw()` to be known only in the context of a `point`.

The `point` structure could be used as follows:

```
point                // two points on
    a,                // screen
    b;

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
```

The function that is part of the structure is selected in a similar manner in which data fields are selected; i.e., using the field selector operator (`.`). When pointers to structs are used, `->` can be used.

The idea of this syntactical 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 function `draw()` could be declared which would draw the circle.



## Chapter 3

# A first impression of C++

In this chapter the usage of C++ is further explored. The possibility to declare functions in `structs` is further illustrated using examples. The concept of a `class` is introduced.

### 3.1 More extensions to C in C++

Before we continue with the 'real' object-oriented approach to programming, we first introduce some extensions to the C programming language, encountered in C++: not mere differences between C and C++, but syntactical constructs and keywords that are not found in C.

#### 3.1.1 The scope resolution operator ::

The syntax of C++ introduces a number of new operators, of which the scope resolution operator `::` is described first. This operator can be used in situations where a global variable exists with the same name as a local variable:

```
#include <stdio.h>

int
    counter = 50;                // global variable

int main()
{
    for (register int counter = 1; // this refers to the
        counter < 10;            // local variable
        counter++)
    {
        printf("%d\n",
            ::counter             // global variable
            /                     // divided by
            counter);            // local variable
    }
    return (0);
}
```

In this code fragment the scope operator is used to address a global variable instead of the local variable with the same name. The scope operator is more extensively used than shown here, but its main purpose will be described later (in chapter 6).

### 3.1.2 'cout', 'cin' and 'cerr'

In analogy to C, C++ defines standard input- and output streams which are opened 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 read from the streams or written to them using the operators `<<`, called the *insertion operator* and `>>`, called the *extraction operator*. This is illustrated in the example below:

```
#include <iostream>

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

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

    cout << "The number is: " << ival << endl
         << "And the string is: " << sval << endl;
}
```

This program reads a number and a string from the `cin` stream (usually the keyboard) and prints these data to `cout`. Concerning the streams and their usage we note:

- The streams are declared in the header file `iostream`. In the examples in the 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 unless otherwise stated.
- The streams `cout`, `cin` and `cerr` are in fact 'objects' of a given class (more on classes later), processing the input and output of a program. Note that the term 'object', as used here, means the set of data and functions which defines the item in question.
- The stream `cin` reads data and copies the information to variables (e.g., `ival` in the above example) using the extraction operator `>>`. We will describe later how operators in C++ can perform quite different actions than what they are defined to do by the language grammar,

such as is the case here. We've seen function overloading. 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 type of supplied variables.
- Special symbolic constants are used for special situations. The termination of a line written by `cout` is realized by inserting the `endl` symbol, rather than using the string `"\n"`.

The streams `cin`, `cout` and `cerr` are in fact not part of the C++ grammar, as defined in the compiler which parses source files. The streams are part of the definitions in the header file `iostream`. This is comparable to the fact that functions as `printf()` are not part of the C grammar, but were originally written by people who considered such functions handy and collected them in a run-time library.

Whether a program uses the old-style functions like `printf()` and `scanf()` or whether it employs the new-style streams is a matter of taste. Both styles can even be mixed. A number of advantages and disadvantages is given below:

- Compared to the standard C functions `printf()` and `scanf()`, the usage of the insertion and extraction operators is more *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.
- The functions `printf()` and `scanf()`, and other functions which use format strings, in fact implement a mini-language which is interpreted at run-time. In contrast, the C++ compiler knows exactly which in- or output action to perform given which argument.
- The usage of the left-shift and right-shift operators in the context of the streams does illustrate the possibilities of C++. Again, it requires a little getting used to, ascending from C, but after that these overloaded operators feel rather comfortably.
- `Iostreams` are *extensible*: functionality can easily be added to the existing functionality, a phenomenon called *inheritance*. Inheritance is discussed in detail in chapter 13.

The *iostream library* has a lot more to offer than just `cin`, `cout` and `cerr`. In chapter 5 *iostreams* will be covered in greater detail. Even though `printf()` and friends can still be used in C++ programs, *iostreams* are practically replacing 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 understood the possibilities of *iostream* objects.

### 3.1.3 The keyword 'const'

The keyword `const` very often occurs in C++ programs, although `const` is also part of the C grammar, in C `const` is much less often used.

The `const` keyword is a modifier which states that the value of a variable or of an argument may not be modified. In the below example an attempt is made to change the value of a variable `ival`, which is not legal:

```
int main()
```

```

{
    int const          // a constant int..
        ival = 3;      // initialized to 3

    ival = 4;          // assignment leads
                        // to an error message

    return 0;
}

```

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

```

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

```

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

```
char const *buf;
```

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

In the declaration

```
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

```
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 not often used. For example, Bjarne Stroustrup states (in [http://www.research.att.com/~bs/bs\\_faq2.html#constplacement](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:*

```

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”).*

Below we'll see an example where applying this simple 'before' placement rule for the keyword `const` produces unexpected (i.e., unwanted) results. Apart from that, the 'idiomatic' before-placement conflicts with the notion of *const functions*, which we will encounter in section 6.2, where the keyword `const` is also written behind the name of the function.

The definition or declaration in which `const` is used should be read from the variable or function identifier back to the type identifier:

“Buf is a `const` pointer to `const` characters”

This rule of thumb is especially handy in cases where confusion may occur. In examples of C++ code, 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:

```
char const *buf;
```

What must remain constant here? According to the sloppy interpretation, the pointer cannot be altered (since `const` precedes the pointer-`*`). In fact, the charvalues are the constant entities here, as will be clear when it is tried to compile the following program:

```
int main()
{
    char const *buf = "hello";

    buf++;                // accepted by the compiler
    *buf = 'u';           // rejected by the compiler

    return 0;
}
```

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

Marshall Cline C++ FAQ 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.*

### 3.1.4 References

In addition to the normal declaration of variables, C++ allows 'references' to be declared as synonyms for variables. A reference to a variable is like an *alias*; the variable and the reference can both be used in statements which affect 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 addresses the same memory location which `int_value` occupies. In the definition of `ref`, the reference operator `&` indicates that `ref` is not itself an integer but a reference to one. The two statements

```
int_value++;           // alternative 1
ref++;                // alternative 2
```

have the same effect, as expected. At some memory location an `int` value is increased by one — whether that location is called `int_value` or `ref` does not matter.

References serve an important function in C++ as a means to pass arguments which can be modified. E.g., in standard C, a function which increases the value of its argument by five but which returns nothing (`void`), needs a pointer parameter:

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

int main()
{
    int
        x;

    increase(&x)             // the address of x is
    return 0;               // passed as argument
}
```

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

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

int main()
{
    int
        x;

    increase(x);             // a reference to x is
    return 0;               // passed as argument
}
```

Actually, references are implemented using pointers. So, references in C++ are just pointers, as far as the compiler is concerned. However, the programmer does not need to know or to bother about levels of indirection.

It can be argued whether code such as the above is clear: the statement `increase(x)` in the `main()` function suggests that not `x` itself but a *copy* is passed. Yet the value of `x` changes because of the way `increase()` is defined.

Our suggestions for the usage of references as arguments to functions are therefore the following:

- In those situations where a called function does not alter its arguments, a copy of the variable can be passed:

```
void some_func(int val)
{
    cout << val << endl;
}

int main()
{
    int
        x;

    some_func(x);    // a copy is passed, so
    return 0;        // x won't be changed
}
```

- When a function changes the value of its argument, the address or a reference can be passed, whichever you prefer:

```
void by_pointer(int *valp)
{
    *valp += 5;
}

void by_reference(int &valr)
{
    valr += 5;
}

int main ()
{
    int
        x;

    by_pointer(&x);    // a pointer is passed
    by_reference(x);   // x is altered by reference
    return 0;         // x might be changed
}
```

- References have an important role in those cases where the argument will not be changed by the function, but where it is desirable to pass a reference to the variable instead of a copy of the whole variable. Such a situation occurs when a large variable, e.g., a struct, is passed as argument, or is returned from the function. In these cases the copying operations tend to become significant factors when the entire structure must be copied, and it is preferred to use references. If the argument isn't changed by the function, or if the caller shouldn't change the returned information, the use of the `const` keyword is appropriate and should be used.

Consider the following example:

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

```

        double
            salary;
};

Person
    person[50];           // database of persons
                        // printperson expects a
void printperson (Person const &p)
{
    // reference to a structure
    // but won't change it
    cout << "Name: " << p.name << endl <<
        "Address: " << p.address << endl;

}

// get a person by indexvalue
Person const &getperson(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(getperson(5));
                        // references, not copies
                        // are passed here

    return 0;
}

```

- Furthermore, it should be noted 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 copy constructor is avoided. We have to postpone this discussion until chapter 7.

References also can lead to extremely 'ugly' code. A function can also return a reference to a variable, as in the following example:

```

int &func()
{
    static int
        value;

    return value;
}

```

This allows the following constructions:

```

func() = 20;
func() += func();

```



It is probably superfluous to note that such constructions should not normally be used. Nonetheless, there are situations where it is useful to return a reference. Even though this is discussed more extensively only later, we have seen an example of this phenomenon at our previous discussion of the *iostreams*. In a statement like `cout << "Hello" << endl;`, the insertion operator returns a reference to `cout`. So, in this statement first the "Hello" is inserted into `cout`, producing a reference to `cout`. Via this reference the `endl` is then inserted in the `cout` object, again producing a reference to `cout`. This latter reference is not further used.

A number of differences between pointers and references is pointed out in the list below:

- A reference cannot exist by itself, i.e., without something to refer to. A declaration of a reference like

```
int &ref;
```

is not allowed; what would `ref` refer to?

- References can, however, 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 to what the return value will refer.
- References may be used as data members of classes. We will return to this usage later.
- In contrast, 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.
- In contrast, 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.

## 3.2 Functions as part of structs

The first chapter described that functions can be part of `structs` (see section 2.5.14). Such functions are called *member functions* or *methods*. This section discusses the actual definition of such functions.

The code fragment below illustrates a `struct` in which data fields for a name and address are present. A function `print()` is included in the `struct` definition:

```
struct person
{
    char
        name [80],
        address [80];
```

```

        void
        print (void);
};

```

The member function `print()` is defined using the structure name (`person`) and the scope resolution operator (`::`):

```

void person::print()
{
    cout << "Name:      " << name << endl
          "Address:    " << address<< endl;
}

```

In the definition of this member function, the function name is preceded by the `struct` name followed by `::`. The code of the function shows how the fields of the `struct` can be addressed without using the type name: in this example the function `print()` prints a variable name. Since `print()` is a part of the `struct person`, the variable name implicitly refers to the same type.

This `struct` could be used as follows:

```

person
    p;

strcpy(p.name, "Karel");
strcpy(p.address, "Rietveldlaan 37");
p.print();

```

The advantage of member functions lies in the fact that the called function can automatically address the data fields of the structure for which it was invoked. As such, in the statement `p.print()` the structure `p` is the 'substrate': the variables `name` and `address` which are used in the code of `print()` refer to the same `struct p`.

## 3.3 Several new data types

In C the following basic data types are available: `void`, `char`, `short`, `int`, `long`, `float` and `double`. C++ extends these five basic types with several extra types: the types `bool`, `wchar_t` and `long double`. The type `long double` is merely a double-long double datatype. Apart from these basic types a standard type `string` is available. The datatypes `bool`, and `wchar_t` are covered in the following sections, the datatype `string` is covered in chapter 4.

### 3.3.1 The 'bool' data type

In C the following basic data types are available: `void`, `char`, `int`, `float` and `double`. C++ extends these five basic types with several extra types. In this section the type `bool` is introduced.

The type `bool` represents boolean (logical) values, for which the (now reserved) values `true` and `false` may be used. Apart from these reserved values, integral values may also be assigned to variables of type `bool`, which are implicitly converted to `true` and `false` according to the following conversion rules (assume `intValue` is an `int`-variable, and `boolValue` is a `bool`-variable):

```

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

    // from bool to int:

    intValue = boolValue ? 1 : 0;

```

Furthermore, when `bool` values are inserted into, e.g., `cout`, then 1 is written for `true` values, and 0 is written for `false` values. Consider the following example:

```

cout << "A true value: " << true << endl
    << "A false value: " << false << endl;

```

The `bool` data type is found in other programming languages as well. **Pascal** has its type `Boolean`, and **Java** has a `boolean` type. Different from these languages, C++'s type `bool` acts like a kind of `int` type: it's 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 neglect 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 generally more intuitively clear than using `int`. Consider the following prototypes:

```

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

```

For the first prototype (1), most people will expect the function to return `true` if the given file-name is the name of an existing file. However, using the second prototype some ambiguity arises: intuitively the return value 1 is appealing, as it leads to constructions like

```

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

```

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

As a rule of thumb we 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 when the function returns. Only when the function returns a meaningful integral value (like the sum of two `int` values), let the function return an `int` value.

### 3.3.2 The 'wchar\_t' data type

The `wchar_t` type is an extension of the `char` basic type, to accomodate *wide* character values, such as the *Unicode* character set. `sizeof(wchar_t)` is 2, allowing for 65,536 different character values.

Note that a programming language like **Java** has a data type `char` that is comparable to C++'s `wchar_t` type, while **Java**'s `byte` data type is comparable to C++'s `char` type. Very convenient...

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

As mentioned previously (see section 2.3), C++ contains special syntactical possibilities to implement data hiding. Data hiding is the ability of one program part to hide its data from other parts; thus avoiding improper addressing or name collisions of data.

C++ has three special keywords which are concerned with data hiding: `private`, `protected` and `public`. These keywords can be inserted in the definition of a `struct`. The keyword `public` defines all subsequent fields of a structure as accessible by all code; the keyword `private` defines all subsequent fields as only accessible by the code which is part of the `struct` (i.e., only accessible for the member functions). The keyword `protected` is discussed in chapter 13, and is, for the time being, beyond the scope of the ongoing discussion.

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

```
struct person
{
    public:
        void
            setname (char const *n),
            setaddress (char const *a),
            print (void);
        char const
            *getname (void),
            *getaddress (void);
    private:
        char
            name [80],
            address [80];
};
```

The data fields `name` and `address` are only accessible for the member functions which are defined in the `struct`: these are the functions `setname()`, `setaddress()` etc.. This property of the data type is given by the fact that the fields `name` and `address` are preceded by the keyword `private`. As an illustration consider the following code fragment:

```
person
    x;

x.setname ("Frank");           // ok, setname() is public
strcpy (x.name, "Knarf");      // error, name is private
```

The concept of data hiding is realized here in the following manner. The actual data of a `struct person` are named only in the structure definition. The data are accessed by the outside world by special functions, which 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 data hiding which is thus realized is illustrated further in figure 3.1.

Also note that the functions `setname()` and `setaddress()` are declared as having a `char const *` argument. This means that the functions will not alter the strings which are supplied as their arguments. In the same vein, the functions `getname()` and `getaddress()` return a `char const *`: the caller may not modify the strings which are pointed to by the return values.

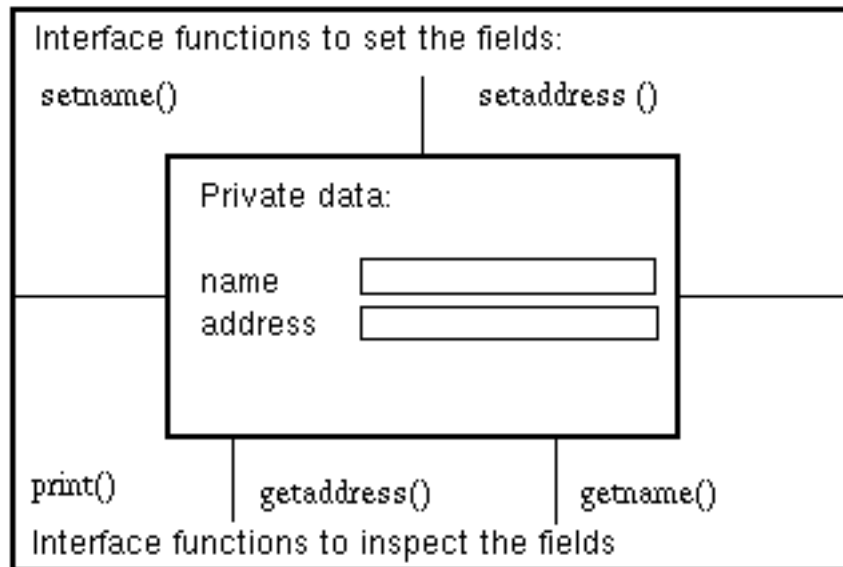


Figure 3.1: Private data and public interface functions of the class `Person`.

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

```
void person::setname(char const *n)
{
    strncpy(name, n, 79);
    name[79] = '\0';
}

char const *person::getname()
{
    return (name);
}
```

In general, the power of the member functions and of the concept of data hiding lies in the fact that the interface functions can perform special tasks, e.g., checks for the validity of data. In the above example `setname()` copies only up to 79 characters from its argument to the data member `name`, thereby avoiding array boundary overflow.

Another example of the concept of data hiding is the following. As an alternative to member functions which keep their data in memory (as do the above code examples), a runtime library could be developed with interface functions which store their data on file. The conversion of a program which stores `person` structures in memory to one that stores the data on disk would mean the relinking of the program with a different library.

Though data hiding can be realized with `structs`, more often (almost always) classes are used instead. A class is in principle equivalent to a `struct` except that unless specified otherwise, all

members (data or functions) are private. As far as private and public are concerned, a class is therefore the opposite of a struct. The definition of a class person would therefore look exactly as shown above, except for the fact that instead of the keyword struct, class would be used. Our typographic suggestion for class names is a capital as first character, followed by the remainder of the name in lower case (e.g., Person).

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

At the end of this chapter we would like to illustrate the analogy between C and C++ as far as structs are concerned. 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. A fragment of an imaginary C header file is given below:

```
// definition of a struct PERSON_
typedef struct
{
    char
        name[80],
        address[80];
} PERSON_;

// some functions to manipulate PERSON_ structs

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

// print information
extern void print(PERSON_ const *p);

// etc..
```

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

```
class Person
{
public:
    void initialize(char const *nm, char const *adr);
    void print(void);
    // etc..
private:
    char
        name[80],
        address[80];
};
```

The struct argument is implicit in C++. A function call in C like

```
PERSON_
x;
```

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

becomes in C++:

```
Person
    x;

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

## 3.6 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 functionname `cos()` is already in use, and that function accepts radians as its arguments, rather than degrees.

Problems like these are normally solved by looking for another name, e.g., the function name `cosDegrees()` is defined. C++ offers an alternative solution by allowing *namespaces* to be defined: areas or regions in the code in which identifiers are defined which cannot conflict with existing names defined elsewhere.

Now that the ANSI/ISO standard is implemented to a large degree in recent compilers, the use of namespaces is more strictly enforced than in previous versions of compilers. This has certain consequences for the setup of class header files. At this point in the Annotations this cannot be discussed in detail, but in section 6.5.1 the construction of header files using entities from namespaces is discussed.

### 3.6.1 Defining namespaces

Namespaces are defined according to the following syntax:

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

The identifier used in the definition of 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 block. So it is not possible to define a namespace within, e.g., a function. However, it is possible to define a namespace using multiple *namespace* declarations. Namespaces are said to be *open*. This means that a namespace `CppAnnotations` could be defined in a file `file1.cc` and also in a file `file2.cc`. The entities defined in the `CppAnnotations` namespace of files `file1.cc` and `file2.cc` are then united in one `CppAnnotations` namespace region. For example:

```
// 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 also be defined outside of their namespaces. This topic is discussed in section 3.6.4.

### 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 of a namespace in a header file which can thereupon be included in sources in which the entities of a namespace are used. Such a header file could contain, e.g.,

```

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

```

### A closed namespace

Namespaces can be defined without a name. Such a namespace is anonymous and it restricts the usability of the defined entities to the source file in which the anonymous namespace is defined.

The entities that are defined in the anonymous namespace are accessible the same way as `static` functions and variables in C. The `static` keyword can still be used in C++, but its use is more dominant in `class` definitions (see chapter 6). In situations where static variables or functions are necessary, the use of the anonymous namespace is preferred.

The anonymous namespace is a closed namespace: it is not possible to add entities to the same anonymous namespace using different sources.

### 3.6.2 Referring to entities

Given a namespace and entities that are defined or declared in it, the scope resolution operator can be used to refer to the entities that are defined in that namespace. For example, to use the function `cos()` defined in the `CppAnnotations` namespace the following code could be used:



```

// assume the CppAnnotations namespace is declared in the
// next header file:
#include <CppAnnotations>

int main()
{
    cout << "The cosine of 60 degrees is: " <<
        CppAnnotations::cos(60) << endl;
    return 0;
}

```

This is a rather cumbersome way to refer to the `cos()` function in the `CppAnnotations` namespace, especially so if the function is frequently used.

Therefore, an *abbreviated* form (just `cos()`) can be used by declaring that `cos()` will refer to `CppAnnotations::cos()`. For this, the *using-declaration* can be used. Following

```

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

```

the function `cos()` will refer to the `cos()` function in the `CppAnnotations` namespace. This implies that the standard `cos()` function, accepting radians, cannot be used automatically anymore. The plain scope resolution operator can be used to reach the generic `cos()` function:

```

int main()
{
    using CppAnnotations::cos;
    ...
    cout << cos(60)           // uses CppAnnotations::cos()
        << ::cos(1.5)        // uses the standard cos() function
        << endl;
    return 0;
}

```

Note that a *using-declaration* can be used inside a block. The *using declaration* prevents the definition of entities having the same name as the one used in the *using declaration*: it is not possible to use a *using declaration* for a variable value in the `CppAnnotations` namespace, and to define (or declare) an identically named object in the block in which the *using declaration* was placed:

```

int main()
{
    using CppAnnotations::value;
    ...
    cout << value << endl; // this uses CppAnnotations::value

    int
        value;           // error: value already defined.

    return 0;
}

```

## The ‘using’ directive

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

```
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 the CppAnnotations namespace (assuming the entities are declared or defined separately from the directive), it is at the same time a somewhat dirty way to do so, as it is less clear which entity will be used in a particular block of code.

If, e.g., `cos()` is defined in the CppAnnotations namespace, the function `CppAnnotations::cos()` will be used when `cos()` is called in the code. 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 which entity will be used as the using declaration does. For this reason, the using directive is somewhat deprecated.

## ‘Koenig lookup’

If *Koenig lookup* were called the ‘Koenig principle’, it could have been the title of a new Ludlum novell. But, unfortunately, it isn’t.

‘Koenig lookup’ refers to the fact that if a function is called without referencing a namespace, then the namespaces of its arguments are used to find the namespace of the function. If a function by the used name is found, then that function is used. This is called the ‘Koenig lookup’.

In the following example this is illustrated. The function `FBB::fun(FBB::Value v)` is defined in the FBB namespace. As shown, it can be called without the explicit mentioning of a namespace:

```
#include <iostream>

namespace FBB
{
    enum Value          // defines FBB::Value
    {
        first,
        second,
    };

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

int main()
{
    fun(FBB::first);    // Koenig lookup: no namespace
                       // for fun()
}
```

```

/*
    generated output:
    fun called for 0
*/

```

Note that trying to fool the compiler doesn't work: if in the namespace FBB `Value` was defined as `typedef int Value` then `FBB::Value` would have been recognized as `int`, thus causing the Koenig lookup to fail.

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

```

#include <iostream>

namespace FBB
{
    enum Value          // defines FBB::Value
    {
        first,
        second,
    };

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

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

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

```

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

```

#include <iostream>

namespace ES
{

```

```

enum Value          // defines ES::Value
{
    first,
    second,
};
}

namespace FBB
{
    enum Value          // defines FBB::Value
    {
        first,
        second,
    };

    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: must be resolved
                                   // by explicitly mentioning
                                   // the namespace
    */
    ES::fun(FBB::first, ES::first);
}
/*
    generated output:
    ES::fun() called
*/

```

### 3.6.3 The standard namespace

Apart from the anonymous namespace, many entities of the runtime available software (e.g., `cout`, `cin`, `cerr` and the templates defined in the *Standard Template Library*, see chapter 17) are now defined in the `std` namespace.

Regarding the discussion in the previous section, one should use a `using` declaration for these entities. For example, in order to use the `cout` stream, the code should start with something like

```
#include <iostream>
```

```
using std::cout;
```

Often, however, the identifiers that are defined in the `std` namespace can all be accepted without much thought. Because of that, one frequently encounters a `using` directive, rather than a `using` declaration with the `std` namespace. So, instead of the mentioned `using` declaration a construction like

```
#include <iostream>

using namespace std;
```

is encountered. Whether this should be encouraged is subject of some dispute. Long `using` declarations are of course inconvenient too. So as a rule of thumb one might decide to stick to `using` declarations, up to the point where the list becomes impractically long, at which point a `using` directive could be considered.

### 3.6.4 Nesting namespaces and namespace aliasing

Namespaces can be nested. The following code shows the definition of a nested namespace:

```
namespace CppAnnotations
{
    namespace Virtual
    {
        void
            *pointer;
    }
}
```

Now the variable `pointer` defined in the `Virtual` namespace, nested under the `CppAnnotations` namespace. In order to refer to this variable, the following options are available:

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

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

- A `using` declaration for `CppAnnotations::Virtual` can be used. Now `Virtual` can be used without any prefix, but `pointer` must be used with the `Virtual::` prefix:

```
...
using CppAnnotations::Virtual;

int main()
{
```

```

        Virtual::pointer = 0;
        return 0;
    }

```

- A using declaration for `CppAnnotations::Virtual::pointer` can be used. Now `pointer` can be used without any prefix:

```

...
using CppAnnotations::Virtual::pointer;

int main()
{
    pointer = 0;
    return 0;
}

```

- A using directive or directives can be used:

```

...
using namespace CppAnnotations::Virtual;

int main()
{
    pointer = 0;
    return 0;
}

```

Alternatively, two separate using directives could have been used:

```

...
using namespace CppAnnotations;
using namespace Virtual;

int main()
{
    pointer = 0;
    return 0;
}

```

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

```

...
using namespace CppAnnotations;
using Virtual::pointer;

int main()
{
    pointer = 0;
    return 0;
}

```

At every using directive all entities of that namespace can be used without any further prefix. If a namespace is nested, then that namespace can also be used without any further prefix. However,

the entities defined in the nested namespace still need the nested namespace's name. Only by using a `using` declaration or directive the qualified name of the nested namespace can be omitted.

When fully qualified names are somehow preferred and a long form like

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

is at the same time considered too long, a *namespace alias* can be used:

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

This defines `CV` as an *alias* for the full name. So, to refer to the pointer variable the construction

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

Of course, a namespace alias itself can also be used in a `using` declaration or directive.

### Defining entities outside of their namespaces

It is not strictly necessary to define members of namespaces within a namespace region. By prefixing the member by its namespace or namespaces a member can be defined outside of a namespace region. This may be done at the global level, or at intermediate levels in the case of nested namespaces. So while it is not possible to define a member of namespace `A` within the region of namespace `C`, it is possible to define a member of namespace `A::B` within the region of namespace `A`.

Note, however, that when a member of a namespace is defined outside of a namespace region, it must *still be declared within* the region.

Assume the type `int INT8[8]` is defined in the `CppAnnotations::Virtual` namespace.

Now suppose we want to define (at the global level) a member function `funny` of namespace `CppAnnotations::Virtual`, returning a pointer to `CppAnnotations::Virtual::INT8`. The definition of such a function could be as follows (first everything is defined inside the `CppAnnotations::Virtual` namespace):

```
namespace CppAnnotations
{
    namespace Virtual
    {
        void
            *pointer;

        typedef int INT8[8];

        INT8 *funny()
        {
            INT8
                *ip = new INT8[1];

            for (int idx = 0; idx < sizeof(INT8) / sizeof(int); ++idx)
                (*ip)[idx] = (1 + idx) * (1 + idx);
        }
    }
}
```

```

        return ip;
    }
}

```

The function `funny()` 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 `funny()` can be defined outside of the `CppAnnotations::Virtual` as follows:

```

namespace CppAnnotations
{
    namespace Virtual
    {
        void
            *pointer;

        typedef int INT8[8];

        INT8 *funny();
    }
}

CppAnnotations::Virtual::INT8 *CppAnnotations::Virtual::funny()
{
    INT8
        *ip = new INT8[1];

    for (int idx = 0; idx < sizeof(INT8) / sizeof(int); ++idx)
    {
        cout << idx << endl;
        (*ip)[idx] = idx * idx;
    }

    return ip;
}

```

At the final code fragment note the following:

- `funny()` 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* of its returntype.
- *Inside* the block of the function `funny` 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 for the function name and its returntype, while the internals of the function would remain the same:

```

namespace CppAnnotations
{

```



```

namespace Virtual
{
    void
        *pointer;

    typedef int INT8[8];

    INT8 *funny();
}

Virtual::INT8 *Virtual::funny()
{
    INT8
        *ip = new INT8[1];

    for (int idx = 0; idx < sizeof(INT8) / sizeof(int); ++idx)
    {
        cout << idx << endl;
        (*ip)[idx] = idx * idx;
    }

    return ip;
}
}

```

## Chapter 4

# The ‘string’ data type

C++ offers a large number of facilities to implement solutions for common problems. Most of these facilities are part of the *Standard Template Library* or they are implemented as *generic algorithms* (see chapter 17).

Among the facilities C++ programmers have developed over and over again (as reflected in the Annotations) are those for manipulating chunks of text, commonly called *strings*. The C programming language offers rudimentary string support: the *ASCII-Z* terminated series of characters is the foundation on which a large amount of code has been built<sup>1</sup>.

Standard C++ now offers a `string` type of its own. In order to use `string`-type objects, the header file `string` must be included in sources.

Actually, `string` objects are *class type* variables, and the `class` is introduced for the first time in chapter 6. However, in order to use a `string`, it is not necessary to know what a class is. In this section the operators that are available for strings and some other operations are discussed. The operations that can be performed on strings take the form

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

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

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

can be used to compare both strings. A function like `compare()`, which is part of the `string`-class is called a *member function*. The `string` class offers a large number of these member functions, as well as extensions of some well-known operators, like the assignment (`=`) and the comparison operator (`==`). These operators and functions are discussed in the following sections.

### 4.1 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* `string::npos` is returned. This value is a (symbolic) value of type `string::size_type`, which is (for all practical purposes) an `int`.

<sup>1</sup> We define an ASCII-Z string as a series of ASCII-characters terminated by the ASCII-character zero (hence -Z), which has the value zero, and should not be confused with character '0', which usually has the value `0x30`

Note that in all operations with strings both `string` objects and `char const *` values and variables can be used.

Some `string`-member functions use *iterators*. Iterators will be covered in section 17.2. The member functions that use iterators are listed in the next section (4.2), they are not further illustrated below.

The following operations can be performed on strings:

- Initialization: String objects can be *initialized*. For the initialization a plain ASCII-Z string, another `string` object, or an implicit initialization can be used. In the example, note that the implicit initialization does not have an argument, and may not use an argument list. Not even empty.

```
#include <string>

int main()
{
    string
        stringOne("Hello World"),    // using plain ascii-Z
        stringTwo(stringOne),         // using another string object
        stringThree;                  // implicit initialization to ""
                                     // do not use: stringThree();

    return 0;
}
```

- Assignment: String objects can be assigned to each other. For this the assignment operator (i.e., the `=` operator) can be used, which accepts both a `string` object and a C-style character-string as its right-hand argument:

```
#include <string>

int main()
{
    string
        stringOne("Hello World"),
        stringTwo;

    stringTwo = stringOne;           // assign stringOne to stringTwo
    stringTwo = "Hello world";      // assign a C-string to StringTwo

    return 0;
}
```

- String to ASCII-Z conversion: In the previous example a standard C-string (an ASCII-Z string) was implicitly converted to a `string`-object. The reverse conversion (converting a `string` object to a standard C-string) is not performed automatically. In order to obtain the C-string that is stored within the `string` object itself, the member function `c_str()`, which returns a `char const *`, can be used:

```
#include <string>

int main()
{
    string
```

```

        stringOne("Hello World");
char const
    *Cstring = stringOne.c_str();

cout << Cstring << endl;

return 0;
}

```

- String elements: The individual elements of a string object can be reached for reading or writing. For this operation the subscript-operator ([]) is available, but there is *no* string pointer dereferencing operator (\*). The subscript operator does not perform range-checking. String range checking is done by the `string::at()` member function:

```

#include <string>

int main()
{
    string
        stringOne("Hello World");

    stringOne[6] = 'w';           // now "Hello world"
    if (stringOne[0] == 'H')
        stringOne[0] = 'h';      // now "hello world"

    // *stringOne = 'H';         // THIS WON'T COMPILE

    stringOne = "Hello World";   // Now using the at()
                                // member function:
    stringOne.at(6) =
        stringOne.at(0);         // now "Hello World"
    if (stringOne.at(0) == 'H')
        stringOne.at(0) = 'W';   // now "Wello World"

    return 0;
}

```

When an illegal index is passed to the `at()` member function, the program aborts.

- Comparisons: Two strings can be compared for (in)equality or ordering, using the `==`, `!=`, `<`, `<=`, `>` and `>=` operators or the `string::compare()` member function. The `compare()` member function comes in several flavors (see section 4.2.4 for details), e.g.:

- `int string::compare(string const &other)`: this variant offers a bit more information than the comparison-operators do. The return value of the `string::compare()` member function may be used for lexicographical ordering: a negative value is returned if the string stored in the string object using the `compare()` member function (in the example: `stringOne`) is located earlier in the *ASCII collating sequence* than the string stored in the string object passed as argument.

```

#include <iostream>
#include <string>

using namespace std;

int main()

```

```

{
    string
        stringOne("Hello World"),
        stringTwo;

    if (stringOne != stringTwo)
        stringTwo = stringOne;

    if (stringOne == stringTwo)
        stringTwo = "Something else";

    if (stringOne.compare(stringTwo) > 0)
        cout <<
            "stringOne after stringTwo in the alphabet\n";
    else if (stringOne.compare(stringTwo) < 0)
        cout <<
            "stringOne before stringTwo in the alphabet\n";
    else
        cout << "Both strings are the same\n";

    // Alternatively:

    if (stringOne > stringTwo)
        cout <<
            "stringOne after stringTwo in the alphabet\n";
    else if (stringOne < stringTwo)
        cout <<
            "stringOne before stringTwo in the alphabet\n";
    else
        cout << "Both strings are the same\n";

    return 0;
}

```

Note that there is no member function to perform a case insensitive comparison of strings.

- `int string::compare(string const &other, string::size_type pos, unsigned n)`: the third argument indicates the number of characters that should be compared. If its value exceeds the number of available characters, only the available characters are compared.
- More variants of `string::compare()` are available. As stated, refer to section 4.2.4 for details.

- **Appending:** A string can be appended to another string. For this the `+=` operator can be used, as well as the `string &string::append()` member function.

Like the `compare()` function, the `append()` member function may have extra arguments. The first argument is the string to be appended, the second argument specifies the index position of the first character that will be appended. The third argument specifies the number of characters that will be appended. If the first argument is of type `char const *`, only a second argument may be specified. In that case, the second argument specifies the number of characters of the first argument that are appended to the `string` object. Furthermore, the `+` operator can be used to append two strings within an expression:

```

#include <iostream>
#include <string>

```

```

using namespace std;

int main()
{
    string
        stringOne("Hello"),
        stringTwo("World");

    stringOne += " " + stringTwo;

    stringOne = "hello";
    stringOne.append(" world");

    stringOne.append(" ok. >This is not used<", 5); // append 5 characters:

    cout << stringOne << endl;

    string
        stringThree("Hello");

    stringThree.append(stringOne, 5, 6); // append " World":

    cout << stringThree << endl;
}

```

The + operator can be used in cases where at least one term of the + operator is a string object (the other term can be a string, char const \* or char).

When neither operand of the + operator is a string, at least one operand must be converted to a string object first. An easy way to do this is to use an *anonymous string object*:

```
string("hello") + " world";
```

- Insertions: The string &string::insert() member function to insert (parts of) a string has at least two, and at most four arguments:
  - The first argument is the offset in the current string object where another string should be inserted.
  - The second argument is the string to be inserted.
  - The third argument specifies the index position of the first character in the provided string-argument that will be inserted.
  - The fourth argument specifies the number of characters that will be inserted.

If the first argument is of type char const \*, the fourth argument is not available. In that case, the third argument indicates the number of characters of the provided char const \* value that will be inserted.

```

#include <string>

int main()
{
    string
        stringOne("Hell ok.");
        // Insert "o " at position 4
    stringOne.insert(4, "o ");
}

```

```

    string
        world("The World of C++");

                                // insert "World" into stringOne
    stringOne.insert(6, world, 4, 5);

    cout << "Guess what ? It is: " << stringOne << endl;
}

```

Several variants of `string::insert()` are available. See section 4.2 for details.

- **Replacements:** At times, the contents of `string` objects must be replaced by other information. To replace parts of the contents of a `string` object by another string the member function `string &string::replace()` can be used. The member function has at least three and possibly five arguments, having the following meanings (see section 4.2 for overloaded versions of `replace()`, using different types of arguments):
  - The first argument indicates the position of the first character that must be replaced
  - The second argument gives the number of characters that must be replaced.
  - The third argument defines the replacement text (a `string` or `char const *`).
  - The fourth argument specifies the index position of the first character in the provided `string`-argument that will be inserted.
  - The fifth argument can be used to specify the number of characters that will be inserted.

If the third argument is of type `char const *`, the fifth argument is not available. In that case, the fourth argument indicates the number of characters of the provided `char const *` value that will be inserted.

The following example shows a very simple *filechanger*: it reads lines from `cin`, and replaces occurrences of a 'searchstring' by a 'replacestring'. Simple tests for the correct number of arguments and the contents of the provided strings (they should be unequal) are implemented using the `assert()` macro.

```

#include <string>
#include <cassert>

int main(int argc, char **argv)
{
    assert(argc == 3 &&
        "Usage: <searchstring> <replacestring> to process stdin");

    string
        line,
        search(argv[1]),
        replace(argv[2]);

    assert(search != replace);

    while (getline(cin, line))
    {
        while (true)
        {
            string::size_type
                idx;

```

```

        idx = line.find(search);

        if (idx == string::npos)
            break;

        line.replace(idx, search.size(), replace);
    }
    cout << line << endl;
}
return 0;
}

```

- Swapping: A particular form of replacement is swapping: the member function `string &string::swap(string &other)` swaps the contents of two `string`-objects. For example:

```

#include <string>

int main()
{
    string
        stringOne("Hello"),
        stringTwo("World");

    cout << "Before: stringOne: " << stringOne << ", stringTwo: "
        << stringTwo << endl;

    stringOne.swap(stringTwo);

    cout << "After: stringOne: " << stringOne << ", stringTwo: "
        << stringTwo << endl;

    return 0;
}

```

- Erasing: The member function `string &string::erase()` removes characters from a `string`. The standard form has two optional arguments:
  - If no arguments are specified, the stored string is erased completely: it becomes the empty string (`string()` or `string("")`).
  - The first argument may be used to specify the offset of the first character that must be erased.
  - The second argument may be used to specify the number of characters that are to be erased.

See section 4.2 for overloaded versions of `erase()`. An example of the use of `erase()` is given below:

```

#include <string>

int main()
{
    string
        stringOne("Hello Cruel World");
}

```



```

        stringOne.erase(5, 6);

        cout << stringOne << endl;

        stringOne.erase();

        cout << "'" << stringOne << "'\n";

        return (0);
    }

```

- **Searching:** To find substrings in a `string` the member function `string::size_type string::find()` can be used. This function looks for the string that is provided as its first argument in the `string` object calling `find()` and returns the index of the first character of the substring if found. If the string is not found `string::npos` is returned. The member function `rfind()` looks for the substring from the end of the `string` object back to its beginning. An example using `find()` was given earlier.
- **Substrings:** To extract a substring from a `string` object, the member function `string string::substr()` is available. The returned `string` object contains a copy of the substring in the `string`-object calling `substr()`. The `substr()` member function has two optional arguments:
  - Without arguments, a copy of the `string` itself is returned.
  - The first argument may be used to specify the offset of the first character to be returned.
  - The second argument may be used to specify the number of characters that are to be returned.

For example:

```

#include <string>

int main()
{
    string
        stringOne("Hello World");

    cout << stringOne.substr(0, 5) << endl
         << stringOne.substr(6)   << endl
         << stringOne.substr()    << endl;
}

```

- **Character set searches:** Whereas `find()` is used to find a substring, the functions `find_first_of()`, `find_first_not_of()`, `find_last_of()` and `find_last_not_of()` can be used to find *sets* of characters (Unfortunately, regular expressions are not supported here). The following program reads a line of text from the standard input stream, and displays the substrings starting at the first vowel, starting at the last vowel, and not starting at the first digit:

```

#include <iostream>
#include <string>

int main()
{
    string
        line;
}

```

```

getline(cin, line);

string::size_type
    pos;

cout << "Line: " << line << endl
    << "Starting at the first vowel:\n"
    << ""
    << (
        (pos = line.find_first_of("aeiouAEIOU"))
        != string::npos ?
            line.substr(pos)
        :
            "*** not found ***"
    ) << ""\n"
    << "Starting at the last vowel:\n"
    << ""
    << (
        (pos = line.find_last_of("aeiouAEIOU"))
        != string::npos ?
            line.substr(pos)
        :
            "*** not found ***"
    ) << ""\n"
    << "Not starting at the first digit:\n"
    << ""
    << (
        (pos = line.find_first_not_of("1234567890"))
        != string::npos ?
            line.substr(pos)
        :
            "*** not found ***"
    ) << ""\n";
}

```

- String size: The number of characters that are stored in a string are obtained by the `size()` member function, which, like the standard C function `strlen()` does not include the terminating ASCII-Z character. For example:

```

#include <string>

int main()
{
    string
        stringOne("Hello World");

    cout << "The length of the stringOne string is "
        << stringOne.size() << " characters\n";

    return 0;
}

```

- Empty strings: The `size()` member function can be used to determine whether a string holds no characters. Alternatively, the `string::empty()` member function can be used:

```

#include <string>

```

```

int main()
{
    string
        stringOne;

    cout << "The length of the stringOne string is "
        << stringOne.size() << " characters\n"
        "It is " << (stringOne.empty() ? "" : " not ")
        << "empty\n";

    stringOne = "";

    cout << "After assigning a \"\"-string to a string-object\n"
        "it is " << (stringOne.empty() ? "also" : " not")
        << " empty\n";

    return 0;
}

```

- Resizing strings: If the size of a string is not enough (or if it is too large), the member function `void string::resize()` can be used to make it longer or shorter. Note that operators like `+=` automatically resize a string when needed.
- Reading a string from a stream: The `istream &getline(istream) tt(&instream, string &target, char delimiter)` member function may be used to read a line of text (up to the first delimiter or the end of the stream) from `instream`.

The delimiter has a default value `'\n'`. It is removed from `instream`, but it is *not* stored in `target`. `Istream::fail()` may be called to determine whether the delimiter was found. If it returns `true` the delimiter was *not* found (see chapter 5 for details about `istream` objects). The function `getline()` was used in several earlier examples (e.g., with the `replace()` member function).

## 4.2 Overview of operations on strings

In this section the available operations on strings are summarized. There are four subparts here: the `string`-initializers, the `string`-iterators, the `string`-operators and the `string`-member functions.

The member functions are ordered alphabetically by the name of the operation. Below, `object` is a `string`-object, and `argument` is either a `string` or a `char const *`, unless overloaded versions tailored to `string` and `char const *` parameters are explicitly mentioned. `Object` is used in cases where a `string` object is initialized or given a new value. `Argument` remains unchanged.

With member functions the types of the parameters are given in a function-prototypical way. With several member functions *iterators* are used. At this point in the Annotations it's a bit premature to discuss iterators, but for referential purposes they have to be mentioned nevertheless. So, a forward reference is used here: see section 17.2 for a more detailed discussion of *iterators*.

Finally, note that all `string`-member functions returning indices in `object` return the predefined constant `string::npos` if no suitable index could be found.

### 4.2.1 The string initializers

The following `string` constructors are available:

- `string object`:  
Initializes `object` to an empty string.
- `string object(string::size_type n, char c)`:  
Initializes `object` with `n` characters `c`.
- `string object(string argument)`:  
Initializes `object` with `argument`.
- `string object(string argument, string::size_type idx, tt(string::size_type n = pos))`:  
Initializes `object` with `argument`, using `n` characters of `argument`, starting at index `idx`.
- `string object(InputIterator begin, InputIterator end)`:  
Initializes `object` with the range of characters implied by the provided `InputIterators`.

### 4.2.2 The string iterators

See section 17.2 for details about *iterators*.

- Forward iterators returned by the members:
  - `string::begin()`
  - `string::end()`
- Reverse iterators returned by the members:
  - `string::rbegin()`
  - `string::rend()`

### 4.2.3 The string operators

The following `string` operators are available:

- `object = argument`.  
Assignment of `argument` to `object`. May also be used for initializing `string` objects.
- `object = c`.  
Assignment of `char c` to `object`. May *not* be used for initializing `string` objects.
- `object += argument`.  
Appends `argument` to `object`. Argument may also be a `char` value.

- `argument1 + argument2`.

Within expressions, strings may be added. At least one term of the expression (the left-hand term or the right-hand term) should be a `string` object. The other term may be a `string`, a `char const *` value or a `char` value, as illustrated by the following example:

```
void fun()
{
    char const
        *asciiz = "hello";
    string
        first = "first",
        second;

    // all expressions compile ok:
    second = first + asciiz;
    second = asciiz + first;
    second = first + 'a';
    second = 'a' + first;
}
```

- `object[string::size_type pos]`.

The subscript-operator may be used to assign individual characters of `object` or to retrieve these characters. There is no range-checking. If range checking is required, use the `at()` member function, summarized earlier.

- `argument1 == argument2`.

The equality operator (`operator==()`) may be used to compare a `string` object to another `string` or `char const *` value. The operator `operator!=()` is available as well. The return value for both is a `bool`. For two identical strings `operator==()` returns `true`, and `operator!=()` returns `false`.

- `argument1 < argument2`.

The less-than operator may be used to compare the ordering within the Ascii-character set of `argument1` and `argument2`. The operators `<=`, `>` and `>=` are available as well.

- `ostream stream; stream << argument`.

The insertion-operator may be used with `string` objects.

- `istream stream; stream >> object`.

The extraction-operator may be used with `string` objects. It operates analogously to the extraction of characters into a character array, but `object` is automatically resized to the required number of characters.

#### 4.2.4 The string member functions

The string member functions are listed in alphabetical order. The member name, prefixed by the `string`-class is given first. Then the full prototype and a description are given. Values of the type `string::size_type` represent index positions within a `string`. For all practical purposes, these

values may be interpreted as `int`. The special value `string::npos` is defined to represent a non-existing index.

- `char &string::at(size_type pos):`

The character (reference) at the indicated position is returned (it may be reassigned). The member function performs range-checking, aborting the program if an invalid index is passed.

- `string &string::append(InputIterator begin, InputIterator end):`

Using this member function the range of characters implied by the `begin` and `end` `InputIterators` are appended to the `string` object.

- `string &string::append(string argument, size_type pos, size_type n):`

- If only `argument` is given, it is appended to the `string` object.
- If `pos` is specified as well, `argument` is appended from index position `pos` until the end of `argument`.
- If all three arguments are provided, `n` characters of `argument`, starting at index position `pos` are appended to the `string` object.

If `argument` is of type `char const *`, parameter `pos` cannot be specified. So, with `char const *` arguments, either *all* characters or an *initial subset* of the characters of the provided `char const *` `argument` are appended to the `string` object.

- `string &string::append(size_type n, char c):`

Using this member function, `n` characters `c` can be appended to the `string` object.

- `string &string::assign(string argument, size_type pos, size_type n):`

- If only `argument` is given, it is assigned to the `string` object.
- If `pos` is specified as well, the `string` object is assigned from index position `pos` until the end of `argument`.
- If all three arguments are provided, `n` characters of `argument`, starting at index position `pos` are assigned to the `string` object.

If `argument` is of type `char const *`, no parameter `pos` is available. So, with `char const *` arguments, either *all* characters or an *initial subset* of the characters of the provided `char const *` `argument` are assigned to the `string` object.

- `string &string::assign(size_type n, char c):` Using this member function, `n` characters `c` can be assigned to the `string` object.

- `size_type string::capacity():`

returns the number of characters that can currently be stored inside the `string` object.

- `int string::compare(string argument):`

This member function can be used to compare (according to the ASCII-character set) the text stored in the `string` object and in `argument`. The `argument` may also be a (non-0) `char const *`. 0 is returned if the characters in the `string` object and in `argument` are the same; a negative value is returned if the text in `string` is lexicographically *before* the text in `argument`; a positive value is returned if the text in `string` is lexicographically *beyond* the text in `argument`.

- `int string::compare(size_type idx, size_type len, string argument):`

This member function can be used to compare a substring of the text stored in the `string` object with the text stored in `argument`. At most `len` characters, starting at offset `idx`, are compared with the text in `argument`. If `idx` is or exceeds the number of characters in the `string` object, an (`out_of_range`) exception is thrown (see chapter 8). The `argument` may also be a (non-0) `char const *`.

- `int string::compare(size_type idx, size_type len, string argument, size_type arg_idx, size_type arg_len):`

This member function can be used to compare a substring of the text stored in the `string` object with a substring of the text stored in `argument`. At most `len` characters of the `string` object, starting at offset `idx`, are compared with at most `arg_len` characters of `argument`, starting at offset `arg_idx`. If `idx` or `arg_idx` is or exceeds the number of characters in their respective `string` objects, an (`out_of_range`) exception is thrown. Note that `argument` *must* also be a `string` object.

- `int string::compare(size_type idx, size_type len, char const *argument, size_type arg_len):`

This member function can be used to compare a substring of the text stored in the `string` object with a substring of the text stored in `argument`. At most `len` characters of the `string` object, starting at offset `idx`, are compared with at most `arg_len` characters of `argument`. If `idx` is or exceeds the number of characters in the `string` object, an (`out_of_range`) exception is thrown. `Argument` must have at least `arg_len` characters. However, the characters may have arbitrary values: the ASCII-Z value has no special meaning.

- `size_type string::copy(char *argument, size_type n, size_type pos):`

If the third argument is omitted, the first `n` characters of the `string` object are copied to `argument`. If the third argument is given, copying starts from element `pos` of the `string` object. Following the copying, no ASCII-Z is appended to the copied string. If `n` exceeds the `string` object's `length()`, at most `length()` characters are copied. The actual number of characters that were copied is returned. By using `string::npos` with the `copy()` member, all characters of the `string` object can be copied. A final ASCII-Z character can be appended to the copied text as follows:

```
buffer[s.copy(buffer, string::npos)] = 0;
```

- `char const *string::c_str():`

the member function returns the contents of the `string` object as an ASCII-Z C-string.

- `char const *string::data():`

returns the raw text stored in the `string` object.

- `bool string::empty():`

returns `true` if the `string` object contains no data.

- `string &string::erase(size_type pos; size_type n):`

This member function can be used to erase (a sub)string of the `string` object. The basic form erases the `string` object completely. The working of other forms of `erase()` depend on the specification of extra arguments:

- If `pos` is specified, the contents of the `string` object are erased from index position `pos` until the end of the `string` object.

- If `pos` and `n` are provided, `n` characters of the `string` object, starting at index position `pos` are erased.
- `iterator string::erase(iterator p):`

The contents of the `string` object are erased until (iterator) position `p`. The iterator `p` is returned.
- `iterator string::erase(iterator f, iterator l):`

The range of characters of the `string` object, implied by the iterators `f` and `l` are erased. The iterator `f` is returned.
- `size_type string::find(string argument, size_type pos):`

Returns the index in the `string` object where `argument` is found. If `pos` is omitted, the search starts at the beginning of the `string` object. If `pos` is provided, it refers to the index in the `string` object where the search for `argument` should start.
- `size_type string::find(char const *argument, size_type pos, size_type n):`

Returns the index in the `string` object where `argument` is found. The parameter `n` indicates the number of characters of `argument` that should be used in the search: it defines a partial string starting at the beginning of `argument`. If omitted, all characters in `argument` are used. The parameter `pos` refers to the index in the `string` object where the search for `argument` should start. If the parameter `pos` is omitted as well, the `string` object is scanned completely.
- `size_type string::find(char c, size_type pos):`

Returns the index in the `string` object where `c` is found. If the argument `pos` is omitted, the search starts at the beginning of the `string` object. If provided, it refers to the index in the `string` object where the search for the `string` object should start.
- `size_type string::find_first_of(string argument, size_type pos):`

Returns the index in the `string` object where any character in `argument` is found. If the argument `pos` is omitted, the search starts at the beginning of the `string` object. If provided, it refers to the index in the `string` object where the search for `argument` should start.
- `size_type string::find_first_of(char const* argument, size_type pos, size_type n):`

Returns the index in the `string` object where a character of `argument` is found, no matter which character. The parameter `n` indicates the number of characters of the `string` object that should be used in the search: it defines a partial string starting at the beginning of the `string` object. If omitted, all characters in the `string` object are used. The parameter `pos` refers to the index in the `string` object where the search for `argument` should start. If the parameter `pos` is omitted as well, the `string` object is scanned completely.
- `size_type string::find_first_of(char c, size_type pos):`

Returns the index in the `string` object where character `c` is found. If the argument `pos` is omitted, the search starts at the beginning of the `string` object. If provided, it refers to the index in the `string` object where the search for `c` should start.



- `size_type string::find_first_not_of(string argument, size_type pos):`  
Returns the index in the `string` object where a character not appearing in `argument` is found. If the argument `pos` is omitted, the search starts at the beginning of the `string` object. If provided, it refers to the index in the `string` object where the search for `argument` should start.
- `size_type string::find_first_not_of(char const *argument, size_type pos, size_type n):`  
Returns the index in the `string` object where any character not appearing in `argument` is found. The parameter `n` indicates the number of characters of the `string` object that should be used in the search: it defines a partial string starting at the beginning of the `string` object. If omitted, all characters in the `string` object are used. The parameter `pos` refers to the index in the `string` object where the search for `argument` should start. If the parameter `pos` is omitted as well, the `string` object is scanned completely.
- `size_type string::find_first_not_of(char c, size_type pos):`  
Returns the index in the `string` object where another character than `c` is found. If the argument `pos` is omitted, the search starts at the beginning of the `string` object. If provided, it refers to the index in the `string` object where the search for `c` should start.
- `size_type string::find_last_of(string argument, size_type pos):`  
Returns the last index in the `string` object where a character in `argument` is found. If the argument `pos` is omitted, the search starts at the beginning of the `string` object. If provided, it refers to the index in the `string` object where the search for `argument` should start.
- `size_type string::find_last_of(char const* argument, size_type pos, size_type n):`  
Returns the last index in the `string` object where a character of `argument` is found. The parameter `n` indicates the number of characters of the `string` object that should be used in the search: it defines a partial string starting at the beginning of the `string` object. If omitted, all characters in the `string` object are used. The parameter `pos` refers to the index in the `string` object where the search for `argument` should start. If the parameter `pos` is omitted as well, the `string` object is scanned completely.
- `size_type string::find_last_of(char c, size_type pos):`  
Returns the last index in the `string` object where character `c` is found. If the argument `pos` is omitted, the search starts at the beginning of the `string` object. If provided, it refers to the index in the `string` object where the search for `c` should start.
- `size_type string::find_last_not_of(string argument, size_type pos):`  
Returns the last index in the `string` object where any character not appearing in `argument` is found. If the argument `pos` is omitted, the search starts at the beginning of the `string` object. If provided, it refers to the index in the `string` object where the search for `argument` should start.
- `size_type string::find_last_not_of(char const *argument, size_type pos, size_type n):`  
Returns the last index in the `string` object where any character not appearing in `argument` is found. The parameter `n` indicates the number of characters of the `string` object that should be used in the search: it defines a partial string starting at the beginning of the `string` object. If omitted, all characters in the `string` object are

used. The parameter `pos` refers to the index in the `string` object where the search for argument should start. If the parameter `pos` is omitted as well, all of the `string` object is scanned.

- `size_type string::find_last_not_of(char c, size_type pos):`

Returns the last index in the `string` object where another character than `c` is found. If the argument `pos` is omitted, the search starts at the beginning of the `string` object. If provided, it refers to the index in the `string` object where the search for `c` should start.

- `istream &getline(istream instream, string object, char delimiter):`

This member function can be used to read a line of text (up to the first delimiter or the end of the stream) from `instream`. The delimiter has a default value `'\n'`. It is removed from `instream`, but it is not stored in the `string` object.

- `string &string::insert(size_type t_pos, string argument, size_type pos; size_type n):`

This member function can be used to insert (a sub)string of `argument` into the `string` object, at the `string` object's index position `t_pos`. The basic form inserts `argument` completely at index `t_pos`. The way other forms of `insert()` work depend on the specification of extra arguments:

- If `pos` is specified, `argument` is inserted from index position `pos` until the end of `argument`.
- If `pos` and `n` are provided, `n` characters of `argument`, starting at index position `pos` are inserted into the `string` object.

If `argument` is of type `char const *`, no parameter `pos` is available. So, with `char const *` arguments, either *all* characters or an *initial subset* of the characters of the provided `char const *` argument are inserted into the `string` object.

- `string &string::insert(size_type t_pos, size_type n, char c):` Using this member function, `n` characters `c` can be inserted to the `string` object.

- `iterator string::insert(iterator p, char c):`

The character `c` is inserted at the (iterator) position `p` in the `string` object. The iterator `p` is returned.

- `iterator string::insert(iterator p, size_type n, char c):`

`N` characters `c` are inserted at the (iterator) position `p` in the `string` object. The iterator `p` is returned.

- `iterator string::insert(iterator p, InputIterator first, InputIterator last):`

The range of characters implied by the `InputIterators` `first` and `last` are inserted at the (iterator) position `p` in the `string` object. The iterator `p` is returned.

- `size_type string::length():`

returns the number of characters stored in the `string` object.

- `size_type string::max_size():`

returns the maximum number of characters that can be stored in the `string` object.

- `string& string::replace(size_type pos1, size_type n1, const string argument, size_type pos2, size_type n2):`

The substring of `n1` characters of the `string` object, starting at position `pos1` is replaced by `argument`. If `n1` is set to 0, the member function *inserts* `argument` into the `string` object.

The basic form uses `argument` completely. The way other forms of `replace()` work depends on the specification of extra arguments:

- If `pos2` is specified, `argument` is inserted from index position `pos2` until the end of `argument`.
- If `pos2` and `n2` are provided, `n2` characters of `argument`, starting at index position `pos2` are inserted into the `string` object.

If `argument` is of type `char const *`, no parameter `pos2` is available. So, with `char const *` arguments, either *all* characters or an *initial subset* of the characters of the provided `char const *` `argument` are replaced in the `string` object.

- `string &string::replace(size_type pos, size_type n1, size_type n2, char c):`

This member function can be used to replace `n1` characters of the `string` object, starting at index position `pos`, by `n2` `c`-characters. The argument `n2` may be omitted, in which case the string to be replaced is replaced by just one character `c`.

- `string& string::replace (iterator i1, iterator i2, string argument):`

Here, the string implied by the iterators `i1` and `i2` are replaced by the string `str`. If `argument` is a `char const *`, an extra argument `n` may be used, specifying the number of characters of `argument` that are used in the replacement.

- `iterator string::replace(iterator f, iterator l, string argument):`

The range of characters of the `string` object, implied by the iterators `f` and `l` are replaced by `argument`. If `argument` is a `char const *`, an extra argument `n` may be used, specifying the number of characters of `argument` that are used in the replacement. The string the `string` object is returned.

- `iterator string::replace(iterator f, iterator l, size_type n, char c):`

The range of characters of the `string` object, implied by the iterators `f` and `l` are replaced by `n` `c`-characters. The iterator `f` is returned.

- `string string::replace(iterator i1, iterator i2, InputIterator j1, InputIterator j2):`

Here the range of characters implied by the iterators `i1` and `i2` is replaced by the range of characters implied by the `InputIterators` `j1` and `j2`.

- `void string::resize(size_type n, char c):`

The string stored in the `string` object is resized to `n` characters. The second argument is optional. If provided and the string is enlarged, the extra characters are initialized to `c`.

- `size_type string::rfind(string argument, size_type pos):`

Returns the index in the `string` object where `argument` is found. Searching proceeds either from the end of the `string` object or from offset `pos` back to the beginning. If the argument `pos` is omitted, searching starts at the end of the `string` object. If `pos` is provided, it refers to the index in the `string` object where the search for `argument` should start.

- `size_type string::rfind(char const *argument, size_type pos, size_type n):`

Returns the index in the `string` object where `argument` is found. Searching proceeds either from the end of the `string` object or from offset `pos` back to the beginning. The parameter `n` indicates the number of characters of `argument` that should be used in the search: it defines a partial string starting at the beginning of `argument`. If omitted, all characters in `argument` are used. If the argument `pos` is omitted as well, searching starts at the end of the `string` object. If `pos` is provided, it refers to the index in the `string` object where the search for (a substring of) `argument` should start.

- `size_type string::rfind(char c, size_type pos):`

Returns the index in the `string` object where `c` is found. Searching proceeds either from the end of the `string` object or from offset `pos` back to the beginning.

- `size_type string::size():`

returns the number of characters stored in the `string` object.

- `string string::substr(size_type pos, size_type n):`

Returns a substring of the `string` object. The parameter `n` may be used to specify the number of characters of `argument` that are returned. The parameter `pos` may be used to specify the index of the first character of `argument` that is returned. Either `n` or both arguments may be omitted.

- `size_type string::swap(string argument):`

swaps the contents of the `string` object and `argument`. In this case, `argument` must be a `string` and cannot be a `char const *`.

## Chapter 5

# The IO-stream Library

As an extension to the standard stream (FILE) approach well known from the C programming language, C++ offers an *input/output* (I/O) library based on `class` concepts.

Earlier (in chapter 3) we've already seen examples of the use of the C++ I/O library, especially the use of the insertion operator (`operator<<()`) and the extraction operator (`operator>>()`). In this chapter we'll cover the library 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 the construction of classes will be covered in the upcoming chapter 6, and *inheritance* only in chapter 13, we think it is well possible to introduce input and output (I/O) facilities well before discussing the technical background of these topics.

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:

```
typedef basic_ios<char> ios;
```

Since C++ defines both the `char` and `wchar_t` types, I/O facilities were developed using the *template* mechanism. As will be further elaborated in chapter 18, this way it was possible to construct generic software, which could be used for both the `char` and `wchar_t` types. So, analogous to the above `typedef` there exists a

```
typedef basic_ios<wchar_t> wios;
```

type definition which can be used for the `wchar_t` type. Due to the type definitions, the `basic_` prefix can be omitted in the Annotations. In the Annotations the emphasis is primarily on the standard 8-bits `char` type.

As a side effect to this implementation it must be stressed that it is *not* anymore correct to declare `iostream` objects using standard forward declarations, like:

```
class ostream;           // now erroneous
```

Instead, sources that must declare `iostream` classes must

```
#include <iosfwd>         // correct way to declare iostream classes
```

Using the C++ I/O library 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 `fprintf()` 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 over the years, 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. C++ adds I/O based on classes to FILE-based I/O, resulting in type safety, extensibility, and a clean design. In the ANSI/ISO standard the intent was to construct architecture independent I/O. Previous implementations of the *iostreams* library did not always comply with the standard, resulting in many extensions to the standard. Software developed earlier may have to be partially rewritten with respect to I/O. This is tough for those who are now forced to modify existing software, but every feature and extension that was available in previous implementations can be reconstructed easily using the ANSI/ISO standard conforming I/O library. Not all of these reimplementations can be covered in this chapter, as most use inheritance and polymorphism, topics that will be covered in chapters 13 and 14, respectively. Selected reimplementations will be provided in chapter 19, and below references to particular sections in that chapter will be given where appropriate.

This chapter is organized as follows (see also figure 5.1):

- The class `ios_base` represents the foundation upon which the *iostreams* I/O library was built. The class `ios` forms the foundation of all I/O operations, and defines, among other things, the facilities for inspecting the state of I/O streams and output formatting.
- The class `ios` was directly derived from `ios_base`. Every class of the I/O library doing input or output is *derived* from this `ios` class, and *inherits* its (and, by implication, `ios_base`'s) capabilities. The reader is urged to keep this feature in mind while reading this chapter. The concept of inheritance is not discussed further here, but rather in chapter 13.

An important function of the class `ios` is to define the communication with the *buffer* that is used by streams. The buffer is a `streambuf` object (or is derived from the class `streambuf`) and is responsible for the actual input and/or output. This means that *iostream* objects do not perform input/output 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 is `ostream`, defining the insertion operator as well as other facilities for writing information to streams. Apart from inserting information in files it is possible to insert information in memory buffers, for which the `ostringstream` class is available. Formatting of the output is to a great extent possible using the facilities defined in the `ios` class, but it is also possible to *insert formatting commands* directly in streams, using *manipulators*. This aspect of C++ output is discussed as well.

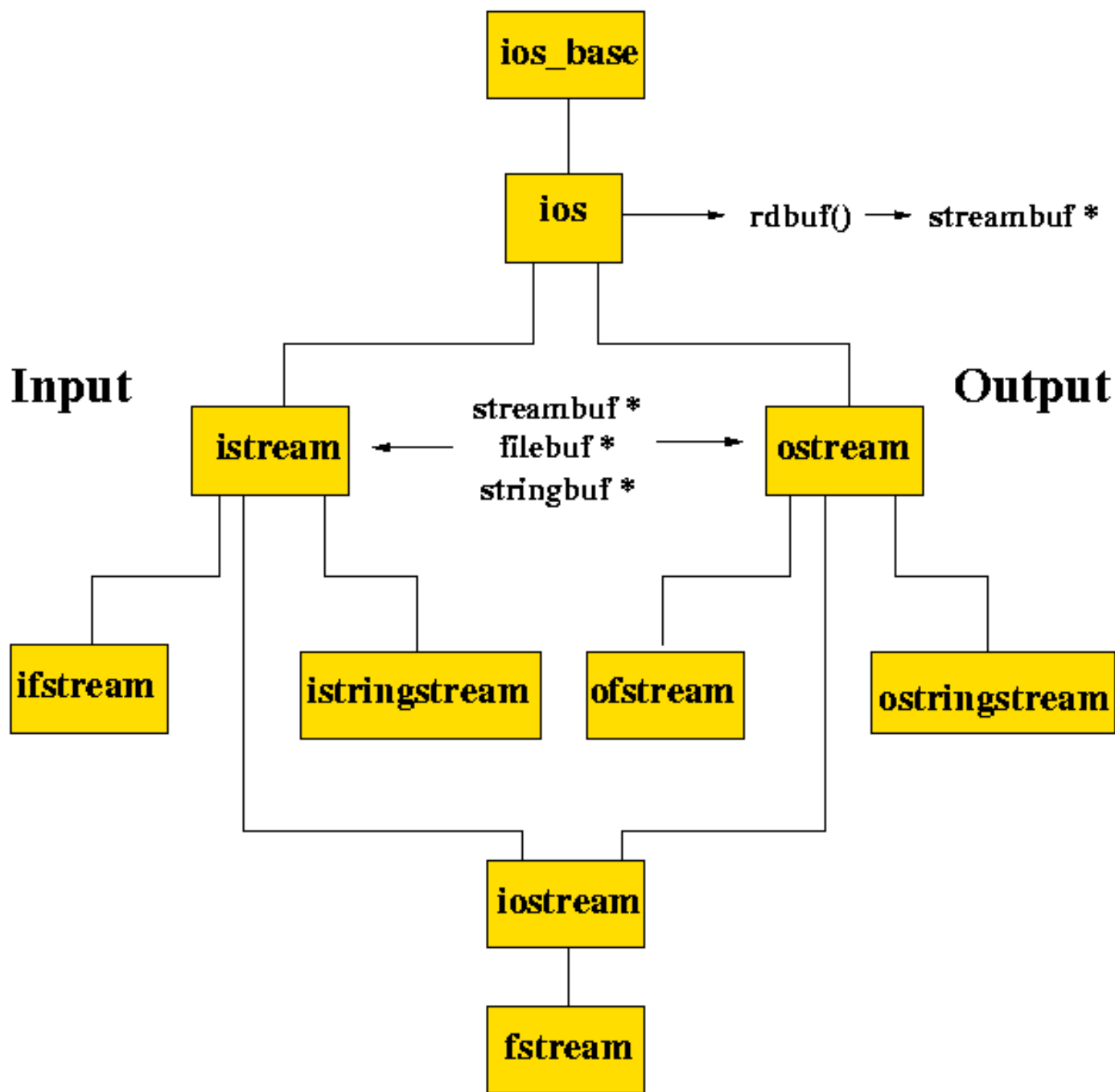


Figure 5.1: Central I/O Classes

- Basic C++ input facilities are available in the `istream` class. This class defines the insertion operator and related facilities for input. Analogous to the `ostream` a class `istream` is available for extracting information from memory buffers.
- Finally, several advanced I/O-related topics are discussed: other topics, combined reading and writing using streams and mixing C and C++ I/O using `filebuf` objects. Other I/O related topics are covered later in the Annotations, e.g., in chapter 19.

In the `iostream` library the stream objects have a limited role: they form 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 for which the `streambuf` object was created in the first place. 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’ `istream`- or `ostream`-class facilities. It is important to understand the distinction between the formatting roles of the `iostream` objects and the buffering interface to an external device as implemented in a `streambuf`. Interfacing to new devices (like *sockets* or *file descriptors*) requires us to construct a new kind of `streambuf`, not a new kind of `istream` or `ostream` object. A *wrapper class* may be constructed around the `istream` or `ostream` classes, though, to ease the access to a special device. This is how the `stringstream` classes were constructed.

## 5.1 `iostream` header files

Several header files are defined for the `iostream` library. Depending on the situation at hand, the following header files should be used:

- `#include <iosfwd>`: sources should use this preprocessor directive if a forward declaration is required for the `iostream` classes. For example, if a function defines a reference parameter to an `ostream` then, when this function itself is declared, there is no need for the compiler to know exactly what an `ostream` is. In the header file declaring such a function the `ostream` class merely needs to be declared. One cannot use

```
class ostream;           // erroneous declaration

void someFunction(ostream &str);
```

but, instead, one should use:

```
#include <iosfwd>        // correctly declares class ostream

void someFunction(ostream &str);
```

- `#include <streambuf>`: sources should use this preprocessor directive when using `streambuf` or `filebuf` classes. See sections 5.7 and 5.7.2.
- `#include <istream>`: sources should use this preprocessor directive when using the class `istream` or when using classes that do both input and output. See section 5.5.1.
- `#include <ostream>`: sources should use this preprocessor directive when using the class `ostream` class or when using classes that do both input and output. See section 5.4.1.
- `#include <iostream>`: sources should use this preprocessor directive when using the global stream objects (like `cin` and `cout`).



- `#include <fstream>`: sources should use this preprocessor directive when using the file stream classes. See sections 5.5.2, 5.4.2 and 5.8.4.
- `#include <sstream>`: sources should use this preprocessor directive when using the string stream classes. See sections 5.4.3 and 5.5.3.
- `#include <iomanip>`: sources should use this preprocessor directive when using parameterized manipulators. See section 5.6

## 5.2 The foundation: the class ‘ios\_base’

The class `ios_base` forms the foundation of all I/O operations, and defines, among other things, the facilities for inspecting the state of I/O streams and most output formatting facilities. Every stream class of the I/O library is, via the class `ios`, *derived* from this class, and *inherits* its capabilities.

The discussion of the class `ios_base` precedes the introduction of members that can be used for actual reading from and writing to streams. But as the `ios_base` class is the basis on which all I/O in C++ was built, we introduce it as the first class of the C++ I/O library.

Note, however, 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 actually does so, and the I/O library represents therefore a *de facto* I/O standard in C++. Also note that, as mentioned before, the `iostream` classes do not do input and output themselves, but delegate this to an auxiliary class: the class `streambuf` or derived from it.

For the sake of completeness and *not* so much because it is necessary to understand the ongoing discussion, it is noted here that it is *not* possible to construct an `ios_base` object directly. As covered by chapter 13, classes that are derived from `ios_base` (like `ios`) may construct `ios_base` objects using the `ios_base::ios_base()` constructor.

The next class in the `iostream` hierarchy (see figure 5.1) is the class `ios`. Since the stream classes inherit from the class `ios`, and thus also from `ios_base`, in practice the distinction between `ios_base` and `ios` is hardly important. Therefore, in the sequel facilities actually provided by `ios_base` are 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`).

## 5.3 Interfacing ‘streambuf’ objects: the class ‘ios’

The `ios` class is directly derived from `ios_base`, and defines *de facto* the foundation for all stream classes of the C++ I/O library.

It is noted that, although it *is* possible to construct an `ios` object directly, this is hardly ever done. The purpose of the class `ios` is to provide the facilities of the class `basic_ios`, and add to this several new facilities, all related to managing 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 offered by the classes `ios` and `ios_base` are also available in the stream classes. Before discussing these stream classes, the facilities offered by the classes `ios_base` and `ios` will now be introduced, pretending they are all facilities provided by the class `ios`.

The class offers several member functions, most of which are related to formatting. Other frequently used member functions are:

- `streambuf *ios::rdbuf();`

This member function returns a pointer to the `streambuf` object forming the interface between the `ios` object and the device with which the `ios` object communicates. See section 19.1.2 for further information about the class `streambuf`.

- `streambuf *ios::rdbuf(streambuf *new);`

This member function can be used to associate a `ios` object 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()`.

- `ostream *ios::tie();`

This member function returns a pointer to the `ostream` object that is currently tied to the `ios` object. The returned `ostream` object is *flushed* every time before information is input or output to the `ios` object of which the `tie()` member is called. The return value 0 indicates that currently no `ostream` object is tied to the `ios` object. See section 5.8.2 for details.

- `ostream *ios::tie(ostream *new);`

This member function can be used to associate an `ios` object with another `ostream` object. A pointer to the `ios` object's original `ostream` object is returned. See section 5.8.2 for details.

### 5.3.1 Condition states

Operations on streams may succeed and they may fail for several reasons. Whenever an operation fails, further read and write operations on the stream are suspended. It is possible to inspect (and possibly: clear) the condition state of streams, so that a program can repair the problem, instead of having to abort.

Conditions are represented by the following *condition flags*:

- `ios::badbit:`

if this flag has been raised an illegal operation has been performed, like an attempt to read beyond end-of-file.

- `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 has failed, like an attempt to extract an `int` when no numeric characters are available on input.

- `ios::goodbit:`

this flag indicates that no failure has been sensed for the `ios` object. The `ios::failbit` and `ios::goodbit` are each other's complements.

Several condition member functions are available to manipulate or determine the states of `ios` objects:

- `ios::bad()`:  
this member function returns a non-zero value when an invalid operation has been requested (i.e., when `ios::goodbit` has been set).
- `ios::eof()`:  
this member function returns a non-zero value when end of file (EOF) has been sensed (i.e., `ios::eofbit` has been set).
- `ios::fail()`:  
this member function returns a non-zero value when `ios::eof()` or `ios::bad()` returns a non-zero value.
- `ios::good()`:  
this member function returns a non-zero value when `ios::fail()` returns a zero value and *vice versa*.

`ios` objects can also be interrogated using *boolean operators*: `true` is returned in boolean expressions for `ios` objects if `ios::good()` would have returned a non-zero value. So, a construction like

```
cin >> x;    // cin is also an ios object

if (cin)
    cout << "Good extraction of 'x'\n";
```

is possible. Also, the complementary test is possible:

```
cin >> x;    // cin is also an ios object

if (!cin)
    cout << "Extraction of 'x' failed\n";
```

Once an error condition has been raised (`ios::goodbit` has *not* been set), further processing of the `ios` object is suspended.

The following members are available for the management of error states:

- `ios::clear()`:  
When an error condition has occurred, and the condition can be repaired, then `clear()` can be called to clear the error status of the file. An overloaded version accepts state flags, which are set after first clearing the current set of flags: `ios::clear(int state)`
- `ios::rdstate()`:  
This member function returns the current set of flags that are set for an `ios` object. To test for a particular flag, use the bitwise and operator:  

```
if (iosObject.rdstate() & ios::good)
{
    // state is good
}
```

- `ios::setstate(int flags):`

This member is used to set a particular set of flags. The member `ios::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 `ios::clear()` is called.

C++ supports an *exception* mechanism for handling exceptional situations. According to the ANSI/ISO standard, exceptions can be used with stream objects. Exceptions are covered in chapter 8. Using exceptions with stream objects is covered in section 8.7.

### 5.3.2 Formatting output and input

The way information is written to streams (or, occasionally, read from streams) may be controlled by *formatting flags*.

Formatting may involve the control of the width of an output field or an input buffer or the form (e.g., the *radix*) in which a value is displayed. Most of the format control is realized in the context of the `ios` class, although most formatting is actually done with output streams, like the upcoming `ostream` class. Since the formatting is controlled by flags, which are defined in the `ios` class, it was considered best to discuss formatting with the `ios` class itself, rather than with a selected derived class, where the choice of the derived class would always be somewhat arbitrarily.

The flags control the formatting, but the flags can be altered basically in two ways: using specialized member functions, discussed in section 5.3.2 or using *manipulators*, which are directly inserted into streams. Manipulators are not applied directly to the `ios` class, as they require the use of the insertion operator. Consequently they are discussed later (in section 5.6).

#### Formatting flags

Most *formatting flags* are related to outputting information. Information can be written to output streams in basically two ways: *binary output* will write information directly to the output stream, without conversion to some human-readable format. E.g., an `int` value is written as a set of four bytes. Alternatively, *formatted output* will convert the values that are stored in bytes in the computer's memory to ASCII-characters, in order to create a human-readable form.

Formatting flags can be used to define the way this conversion takes place, to control, e.g., the number of characters that are written to the output stream.

The following formatting flags are available (see also sections 5.3.2 and 5.6):

- `ios::adjustfield:`

*mask value* used in combination with a flag setting defining the way values are adjusted in wide fields (`ios::left`, `ios::right`, `ios::internal`).

- `ios::basefield:`

*mask value* used in combination with a flag setting the radix of integral values to output (`ios::dec`, `ios::hex` or `ios::oct`).

- `ios::boolalpha:`  
to display boolean values as text, using the text “true” for the true logical value, and the string “false” for the false logical value. By default this flag is not set. Corresponding manipulators: `ios::boolalpha`, `ios::noboolalpha`.
- `ios::dec:`  
to read and display integral values as decimal (i.e., radix 10) values. This is the default. With `setf()` the mask value `ios::basefield` must be provided. Corresponding manipulator: `ios::dec`.
- `ios::fixed:`  
to display real values in a fixed notation (e.g., 12.25), as opposed to displaying values in a scientific notation. With `setf()` the mask value `ios::floatfield` must be provided. Corresponding manipulator: `ios::fixed`.
- `ios::floatfield:`  
mask value used in combination with a flag setting the way real numbers are displayed (`ios::fixed` or `ios::scientific`).
- `ios::hex:`  
to read and display integral values as hexadecimal values (i.e., radix 16) values. With `setf()` the mask value `ios::basefield` must be provided. Corresponding manipulator: `ios::hex`.
- `ios::internal:`  
to add fill characters (blanks by default) between the minus sign of negative numbers and the value itself. With `setf()` the mask value `adjustfield` must be provided. Corresponding manipulator: `ios::internal`.
- `ios::left:`  
to left-adjust (integral) values in fields that are wider than needed to display the values. By default values are right-adjusted (see below). With `setf()` the mask value `adjustfield` must be provided. Corresponding manipulator: `ios::left`.
- `ios::oct:`  
to display integral values as octal values (i.e., radix 8) values. With `setf()` the mask value `ios::basefield` must be provided. Corresponding manipulator: `ios::oct`.
- `ios::right:`  
to right-adjust (integral) values in fields that are wider than needed to display the values. This is the default adjustment. With `setf()` the mask value `adjustfield` must be provided. Corresponding manipulator: `ios::right`.
- `ios::scientific:`  
to display real values in *scientific notation* (e.g., 1.24e+03). With `setf()` the mask value `ios::floatfield` must be provided. Corresponding manipulator: `ios::scientific`.
- `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. Corresponding manipulators: `ios::showbase` and `ios::noshowbase`.

- `ios::showpoint:`

display a trailing decimal point and trailing decimal zeros when real numbers are displayed. When this flag is set, an insertion like:

```
cout << 16.0 << ", " << 16.1 << ", " << 16 << endl;
```

could result in:

```
16.0000, 16.1000, 16
```

Note that the last 16 is an integral rather than a real number, and is not given a decimal point: `ios::showpoint` has no effect here. If `ios::showpoint` is not used, then trailing zeros are discarded. If the decimal part is zero, then the decimal point is discarded as well. Corresponding manipulator: `ios::showpoint`.

- `ios::showpos:`

display a + character with positive values. Corresponding manipulator: `ios::showpos`.

- `ios::skipws:`

used for extracting information from streams. When this flag is is (which is the default) leading white space characters (blanks, tabs, newlines, etc.) are skipped when a value is extracted from a stream. If the flag is not set, leading white space characters are not skipped.

- `ios::unitbuf:`

flush the stream after each output operation.

- `ios::uppercase:`

use capital letters in the representation of (hexadecimal or scientifically formatted) values.

## Format modifying member functions

Several *member functions* are available for I/O formatting. Often, corresponding *manipulators* exist, which may directly be inserted into or extracted from streams using insertion or extraction operators. See section 5.6 for a discussion of the available manipulators. They are:

- `ios &copyfmt(ios &obj):`

This member function copies all format definitions from `obj` to the current `ios` object.

- `ios::fill() const:`

returns (as `char`) the current padding character. By default, this is the blank space.

- `ios::fill(char padding):`

redefines the padding character. Returns (as `char`) the *previous* padding character. Corresponding manipulator: `ios::setfill()`.

- `ios::flags() const`:  
returns the current collection of flags controlling the format state of the stream for which the member function is called. To inspect a particular flag, use the binary and operator, e.g.,  

```
if (cout.flags() & ios::hex)
{
    // hexadecimal output of integral values
}
```
- `ios::flags(fmtflags flagset)`:  
returns the *previous* set of flags, and defines the current set of flags as `flagset`, defined by a combination of formatting flags, combined by the binary or operator.
- `ios::precision() const`:  
returns (as int) the number of significant digits used for outputting real values (default: 6).
- `ios::precision(int signif)`:  
redefines the number of significant digits used for outputting real values, returns (as int) the previously used number of significant digits. Corresponding manipulator: `ios::setprecision()`.
- `ios::setf(fmtflags flags)`:  
returns the *previous* set of *all* flags, and sets one or more formatting flags (using the bitwise operator `|()` to combine multiple flags. Other flags are not affected). Corresponding manipulators: `ios::setiosflags` and `ios::resetiosflags`
- `ios::setf(fmtflags flags, fmtflags mask)`:  
returns the *previous* set of *all* flags, clears all flags mentioned in `mask`, and sets the flags specified in `flags`. Well-known mask values are `ios::adjustfield`, `ios::basefield` and `ios::floatfield`. For example:
  - `setf(ios::left, ios::adjustfield)` is used to left-adjust wide values in their field. (alternatively, `ios::right` and `ios::internal` can be used).
  - `setf(ios::hex, ios::basefield)` is used to activate the hexadecimal representation of integral values (alternatively, `ios::dec` and `ios::oct` can be used).
  - `setf(ios::fixed, ios::floatfield)` is used to activate the fixed value representation of real values (alternatively, `ios::scientific` can be used).
- `ios::unsetf(fmtflags flags)`:  
returns the *previous* set of *all* flags, and clears the specified formatting flags (leaving the remaining flags unaltered). The unsetting of an active default flag (e.g., `cout.unsetf(ios::dec)`) has no effect.
- `ios::width() const`:  
returns (as int) the current output field width (the number of characters to write for numerical values on the next insertion operation). Default: 0, meaning 'as many characters as needed to write the value'. Corresponding manipulator: `ios::setw()`.
- `ios::width(int nchars)`:  
returns (as int) the previously used output field width, redefines the value to `nchars` for the next insertion operation. Note that the field width is reset to 0 after every insertion operation, and that `width()` currently has no effect on text-values like `char*` or `string` values.

## 5.4 Output

Output in C++ evolves around the `ostream` class. The `ostream` class defines the basic operators and members for inserting information into streams: the *insertion operator* (`operator<<()`), and special members like `ostream::write()` for writing unformatted information from streams.

From the class `ostream` several other classes are derived, all having the functionality of the `ostream` class, and adding their own specialties. In the next sections on 'output' we will introduce:

- The class `ostream`, offering the basic facilities for doing output;
- The class `ofstream`, allowing us to open files for writing (comparable to C's `fopen(filename, "w")`);
- The class `ostringstream`, allowing us to write information to memory rather than to files (streams) (comparable to C's `sprintf()` function).

### 5.4.1 Basic output: the class 'ostream'

The class `ostream` is the class defining basic output facilities. The `cout`, `clog` and `cerr` objects are all `ostream` objects. Note that all facilities defined in the `ios` class, as far as output is concerned, is available in the `ostream` class as well, due to the inheritance mechanism (discussed in chapter 13).

`Ostream` objects can be constructed using the following *ostream constructor*:

- `ostream object(streambuf *sb):`  
this constructor can be used to construct a wrapper around an existing open stream, based on an existing `streambuf`, which may be the interface to an existing file. See chapter 19 for examples.

In order to use the `ostream` class in C++ sources, the `#include <ostream>` preprocessor directive must be given. To use the predefined `ostream` objects, the `#include <iostream>` preprocessor directive must be given.

#### Writing to 'ostream' objects

The class `ostream` supports both formatted and *binary output*.

The *insertion operator* (`operator<<()`) may be 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.

Note that the insertion operator points to the `ostream` object wherein the information must be inserted. The normal associativity of `operator<<()` remains unaltered, so when a statement like

```
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

```
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.. For every part of the information that is inserted into the stream the operator returns the `ostream` object into which the information so far was inserted, and the next part of the information to be inserted is devoured.

Streams do not have facilities for formatted output like C's `form()` and `vform()` functions. However, it is not difficult to make these facilities available in the world of streams. In section 19.2 the construction of a class defining a `form()` member function is illustrated.

When binary files must be written, the information should normally not be formatted: an `int` value should be written as a series of unaltered bytes, not as a series of ASCII numeric characters 0 to 9. The following member functions are defined for `ostream` objects:

- `ostream& ostream::put(char c):`

This member function writes 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& ostream::write(char const *buffer, int length):`

This member function writes at most `len` bytes, stored in the `char const *buffer` to the `ostream` object. The bytes are written as they are stored in the buffer, no formatting is done whatsoever.

### '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 randomly.

The following members are available:

- `pos_type ostream::tellp():`

this function returns the current (absolute) position where the next write-operation to the stream will take place. For all practical purposes a `pos_type` can be considered to be an unsigned `long`.

- `ostream &ostream::seekp(off_type step, ios::seekdir org):`

This member function can be used to reposition the stream. The function expects an `off_type step`, the stepsize in bytes to go from `org`. For all practical purposes a `off_type` can be considered to be a `long`. The origin of the step, `org` is an `ios::seekdir` value. Possible values are:

- `ios::beg:`

`org` is interpreted as the stepsize relative to the beginning of the stream. If `org` is not specified, `ios::beg` is used.

- `ios::cur:`

`org` is interpreted as the stepsize relative to the current position (as returned by `tellp()` of the stream).

- `ios::end:`

`org` is interpreted as the stepsize relative to the current end position of the stream.

It is allowed to seek beyond end of file. Writing bytes to a location beyond EOF will pad the intermediate bytes with ASCII-Z values: null-bytes. It is *not* allowed to seek before begin of file. Seeking before `ios::beg` will cause the `ios::fail` flag to be set.

### ‘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 up during the write-operations, and when full it is flushed.

The internal buffer can be flushed under program control:

- `ostream& ostream::flush():`

this member function writes any buffered information to the `ostream` object. The call to `flush()` is implied when:

- The `ostream` object ceases to exist,
- The `endl` or `flush` *manipulators* (see section 5.6) are inserted into the `ostream` object,
- A stream derived from `ostream` (like `ofstream`, see section 5.4.2) is closed.

### 5.4.2 Output to files: the class ‘ofstream’

The `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 preprocessor directive `#include <fstream>` must be given. After including `fstream` `cin`, `cout` etc. are not automatically declared. If these latter objects are needed too, then `iostream` should be included.

The following constructors are available for `ofstream` objects:

- `ofstream` object:

This is the basic constructor. It creates an `ofstream` object which may be associated with an actual file later, using the `open()` member (see below).

- `ofstream` object(`char const *name`, `int mode`):

This constructor can be used to associate an `ofstream` object with the file named `name`, using output mode `mode`. The *output mode* is by default `ios::out`. See section 5.4.2 for a complete overview of available output modes.

In the following example an `ofstream` object, associated with the newly created file `/tmp/scratch`, is constructed:

```
ofstream out("/tmp/scratch");
```

Note that it is not possible to open a `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 `streambuf` object (and in some implementations: with a `filebuf` object, which is also a `streambuf`). `Streambuf` objects are discussed in section 5.7, `filebuf` objects are discussed in section 5.7.2.

Instead of directly associating an `ofstream` object with a file, the object can be constructed first, and opened later.

- `void ofstream::open(char const *name, int mode, int perm):`

Having constructed an `ofstream` object, the member function `open()` can be used to associate the `ofstream` object with an actual file.

- `ofstream::close():`

Conversely, it is possible to close an `ofstream` object explicitly using the `close()` member function. The function sets the `ios::fail` flag of the closed object. Closing the file will flush any buffered information to the associated file. A file is automatically closed when the associated `ofstream` object ceases to exist.

A subtlety is the following: Assume a stream is constructed, but it is not actually 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 will be returned. The 'good' status here indicates that the stream object has been properly constructed. It doesn't mean the file is also open. To test whether a stream is actually open, inspect `ofstream::is_open()`: If `true`, the stream is open. See the following example:

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    ofstream
        of;

    cout << "of's open state: " << boolalpha << of.is_open() << endl;

    of.open("/dev/null");    // on Unix systems

    cout << "of's open state: " << of.is_open() << endl;
}
/*
    Generated output:
    of's open state: false
    of's open state: true
*/
```

## Modes for opening stream objects

The following file modes or file flags are defined for constructing or opening `ofstream` (or `istream`, see section 5.5.2) objects. The values are of type `ios::openmode`:

- `ios::app`:  
reposition to the end of the file before every output command. The existing contents of the file are kept.
- `ios::ate`:  
Start initially at the end of the file. The existing contents of the file are kept.
- `ios::binary`:  
open a binary file (used on systems which make a distinction between text- and binary files, like MS-DOS or MS-Windows).
- `ios::in`:  
open the file for reading. The file must exist.
- `ios::out`:  
open the file. 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:

<code>out   app</code> :	The file is created if non-existing, information is always added to the end of the stream;
<code>out   trunc</code> :	The file is (re)created empty to be written;
<code>in   out</code> :	The stream may be read and written. However, the file must exist.
<code>in   out   trunc</code> :	The stream may be read and written. It is (re)created empty first.

### 5.4.3 Output to memory: the class 'ostringstream'

In order to write information to memory, using the stream facilities, `ostringstream` objects can be used. These objects are derived from `ostream` objects. The following constructors and members are available:

- `ostringstream ostr(string const &s, ios::openmode mode)`:  
When using this constructor, the last or both arguments may be omitted. There is also a constructor available having only an `openmode` parameter. If `string s` is specified and `openmode` is `ios::ate`, the `ostringstream` object is initialized with the `string s` and remaining insertions are appended to the contents of the `ostringstream` object. If `string s` is provided, it will not be altered, as any information inserted into the object is stored in dynamically allocated memory which is deleted when the `ostringstream` object goes out of scope.
- `string ostringstream::str() const`:  
This member function will return the string that is stored inside the `ostringstream` object.

Before the `stringstream` class was available the class `ostrstream` was commonly used for doing output to memory. This latter class suffered from the fact that, once its contents were retrieved using its `str()` member function, these contents were 'frozen', meaning that its dynamically allocated memory was not released when the object went out of scope. Although this situation could be prevented (using the `ostrstream` member call `freeze(0)`), this implementation could easily lead to *memory leaks*. The `stringstream` class does not suffer from these risks. Therefore, the use of the class `ostrstream` is now deprecated in favor of `ostringstream` and `ostrstream` objects should be avoided.

The following example illustrates the use of the `ostringstream` class: several values are inserted into the object. Then, the stored text is stored in a string, whose length and contents are thereupon printed. `Ostringstream` objects are most often used for doing 'type to string' conversions, like converting `int` to `string`. Formatting commands can be used with `stringstreams` as well, as they are available in `ostream` objects. It should also be possible to perform `seek` operations with `ostringstream` objects, but the current Gnu `g++` (version 3.0) implementation apparently does not support them. Here is an example showing the use of an `ostringstream` object:

```
#include <iostream>
#include <string>
#include <sstream>
#include <fstream>

using namespace std;

int main()
{
    ostringstream
        ostr("hello ", ios::ate);

    cout << ostr.str() << endl;

    ostr.setf(ios::showbase);
    ostr.setf(ios::hex, ios::basefield);
    ostr << 12345;

    cout << ostr.str() << endl;

    ostr << " -- ";
    ostr.unsetf(ios::hex);
    ostr << 12;

    cout << ostr.str() << endl;
}
/*
    Output from this program:
hello
hello 0x3039
hello 0x3039 -- 12
*/
```

## 5.5 Input

Input in C++ evolves around the `istream` class. The `istream` class defines the basic operators and members for extracting information from streams: the *extraction operator* (`operator>>()`), and special members like `istream::read()` for reading unformatted information from streams.

From the class `istream` several other classes are derived, all having the functionality of the `istream` class, and adding their own specialties. In the next sections we will introduce:

- The class `istream`, offering the basic facilities for doing input;
- The class `ifstream`, allowing us to open files for reading (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).

### 5.5.1 Basic input: the class 'istream'

The class `istream` is the I/O class defining basic input facilities. The `cin` object is an `istream` object that is declared when sources contain the preprocessor directive `#include <iostream>`. Note that all facilities defined in the `ios` class are, as far as input is concerned, available in the `istream` class as well due to the inheritance mechanism (discussed in chapter 13).

`Istream` objects can be constructed using the following *istream constructor*:

- `istream object(streambuf *sb);`  
this constructor can be used to construct a wrapper around an existing open stream, based on an existing `streambuf`, which may be the interface to an existing file. See chapter 19 for examples.

In order to use the `istream` class in C++ sources, the `#include <istream>` preprocessor directive must be given. To use the predefined `istream` object `cin`, the `#include <iostream>` preprocessor directive must be given.

#### Reading from 'istream' objects

The class `istream` supports both formatted and unformatted *binary input*. The *extraction operator* (`operator>>()`) may be 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 which are stored in the computer's memory.

Note that the extraction operator points to the objects or variables which must receive new values. The normal associativity of `operator>>()` remains unaltered, so when a statement like

```
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

```
cin >> y
```

and the `y` variable is extracted from `cin`.

The `>>` operator has a lot of (overloaded) variants, so many types of variables can be extracted from `istream` objects. There is an overloaded operator `>>()` 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 will (by default) skip all white space characters, and will then extract all consecutive non-white space characters. After processing an extraction operator, the `istream` object into which the information so far was inserted is returned, which will thereupon be used as the *lvalue* for the remaining part of the statement.

Streams do not have facilities for formatted output like C's `scanf()` and `vscanf()` functions. However, it is not difficult to make these facilities available in the world of streams. In section 19.2 the construction of a class defining a `form()` member function is illustrated. The construction of a comparable `iscanstream` class may proceed analogously.

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 istream::gcount():`

this function does not actually read from the input stream, but returns the number of characters that were read from the input stream during the last unformatted input operation.

- `int istream::get():`

this function returns EOF or reads and returns the next available single character as an `int` value.

- `istream& istream::get(char &c):`

this function reads the next single character from the input stream into `c`. As its return value is the stream itself, its return value can be queried to determine whether the extraction succeeded or not.

- `istream& istream::get(char *buffer, int len [, char delim]):`

This function reads a series of `len - 1` characters from the input stream into the array starting at `buffer`, which should be at least `len` bytes long. At most `len - 1` characters are read into the buffer. By default, the delimiter is a newline (`'\n'`) character. The delimiter itself is *not removed* from the input stream.

After reading the series of characters into `buffer`, an ASCII-Z character is written beyond the last character that was written to `buffer`. The functions `eof()` and `fail()` (see section 5.3.1) return 0 (`false`) if the delimiter was not encountered before `len - 1` characters were read. Furthermore, an ASCII-Z can be used for the delimiter: this way strings terminating in ASCII-Z characters may be read from a (binary) file. The program using this `get()` member function should know in advance the maximum number of characters that are going to be read.

- `istream& istream::getline(char *buffer, int len [, char delim]):`

This function operates analogously to the previous `get()` member function, but `delim` is removed from the stream if it is actually encountered. At most `len - 1` bytes are written into the `buffer`, and a trailing ASCII-Z character is appended to the string that was read. The delimiter itself is *not* stored in the `buffer`. If `delim` was *not* found (before reading `len - 1` characters) the `fail()` member function, and possibly also `eof()` will return true.

- `istream& istream::ignore(int n , int delim):`

This member function has two (optional) arguments. When called without arguments, one character is skipped from the input stream. When called with one argument, `n` characters are skipped. The optional second argument specifies a delimiter: after skipping `n` or the `delim` character (whichever comes first) the function returns.

- `int istream::peek():`

this function returns the next available input character, but does not actually remove the character from the input stream.

- `istream& istream::putback (char c):`

The character `c` that was last read from the stream is 'pushed back' into the input stream, to be read again as the next character. EOF is returned if this is not allowed. Normally, one character may always be put back. Note that `c` *must* be the character that was last read from the stream. Trying to put back any other character will fail.

- `istream& istream::read(char *buffer, int len):`

This function reads at most `len` bytes from the input stream into the buffer. If EOF is encountered first, fewer bytes are read, and the member function `eof()` will return true. This function will normally be used for reading *binary* files. Section 5.5.2 contains an example in which this member function is used. The member function `gcount()` should be used to determine the number of characters that were retrieved by the `read()` member function.

- `istream& istream::readsome(char *buffer, int len):`

This function reads at most `len` bytes from the input stream into the buffer. All available characters are read into the buffer, but if EOF is encountered first, fewer bytes are read, without setting the `ios_base::eofbit` or `ios_base::failbit`.

- `istream& istream::unget():`

an attempt is made to push back the last character that was read into the stream. Normally, this succeeds if requested only once after a read operation, as is the case with `putback()`

## 'istream' positioning

Although not every `istream` object supports repositioning, some 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 randomly.

The following members are available:

- `pos_type istream::tellg():`

this function returns the current (absolute) position where the next write-operation to the stream will take place. For all practical purposes a `pos_type` can be considered to be an unsigned `long`.

- `istream &istream::seekg(off_type step, ios::seekdir org):`

This member function can be used to reposition the stream. The function expects a `off_type` `step`, the stepsize in bytes to go from `org`. For all practical purposes a



`pos_type` can be considered to be a `long`. The origin of the step, `org` is a `ios::seekdir` value. Possible values are:

- `ios::beg`:  
    `org` is interpreted as the stepsize relative to the beginning of the stream.  
    If `org` is not specified, `ios::beg` is used.
- `ios::cur`:  
    `org` is interpreted as the stepsize relative to the current position (as returned by `tellg()` of the stream).
- `ios::end`:  
    `org` is interpreted as the stepsize relative to the current end position of the the stream.

While it is allowed to seek beyond end of file, reading at that point will of course fail. It is *not* allowed to seek before begin of file. Seeking before `ios::beg` will cause the `ios::fail` flag to be set.

### 5.5.2 Input from streams: the class ‘`ifstream`’

The class `ifstream` is derived from the class `istream`: it has the same capabilities as the `istream` class, but can be used to access files for reading. Such files must exist.

In order to use the `ifstream` class in C++ sources, the preprocessor directive `#include <fstream>` must be given.

The following constructors are available for `ifstream` objects:

- `ifstream` object:

This is the basic constructor. It creates an `ifstream` object which may be associated with an actual file later, using the `open()` member (see below).

- `ifstream` object(`char const *name`, `int mode`):

This constructor can be used to associate an `ifstream` object with the file named `name`, using input mode `mode`. The *input mode* is by default `ios::in`. See also section 5.4.2 for an overview of available file modes.

In the following example an `ifstream` object is opened for reading. The file must exist:

```
ifstream in("/tmp/scratch");
```

Instead of directly associating an `ifstream` object with a file, the object can be constructed first, and opened later.

- `tt(void ifstream::open(char const *name, int mode, int perm)):`

Having constructed an `ifstream` object, the member function `open()` can be used to associate the `ifstream` object with an actual file.

- `ifstream::close():`

Conversely, it is possible to close an `ifstream` object explicitly using the `close()` member function. The function sets the `ios::fail` flag of the closed object. A file is automatically closed when the associated `ifstream` object ceases to exist.

A subtlety is the following: Assume a stream is constructed, but it is not actually 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 will be returned. The 'good' status here indicates that the stream object has been properly constructed. It doesn't mean the file is also open. To test whether a stream is actually open, inspect `ifstream::is_open()`: If true, the stream is open. See also the example in section 5.4.2.

To illustrate reading from a binary file (see also section 5.5.1), a `double` value is read in binary form from a file in the next example:

```
#include <fstream>
using namespace std;

int main(int argc, char **argv)
{
    ifstream
        f(argv[1]);
    double
        d;

    // reads double in binary form.
    f.read(reinterpret_cast<char *>(&d), sizeof(double));
}
```

### 5.5.3 Input from memory: the class 'istringstream'

In order to read information from memory, using the stream facilities, `istringstream` objects can be used. These objects are derived from `istream` objects. The following constructors and members are available:

- `istringstream istr;`

The constructor will construct an empty `istringstream` object. The object may be filled with information to be retrieved later.

- `istringstream istr(string const &text);`

The constructor will construct an `istringstream` object initialized with the contents of the string `text`.

- `void istringstream::str(string const &text);`

This member function will store the contents of the string `text` into the `istringstream` object, overwriting its current contents.

The `istringstream` object is commonly used for converting ASCII text to its binary equivalent, like the C function `atoi()`. The following example illustrates the use of the `istringstream` class, note especially the use of the member `seekg()`:

```
#include <iostream>
#include <string>
#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 << endl;           // write it out
    istr.seekg(0);               // retry from the beginning
    istr >> x;                   // extract int
    cout << x << endl;           // write it out
    istr.str("666");             // store another text
    istr >> x;                   // extract it
    cout << x << endl;           // write it out
}
/*
    output of this program:
3
123
666
*/

```

## 5.6 Manipulators

`ios` objects define a set of *format flags* that are used for determining the way values are inserted (see section 5.3.2). The format flags can be controlled by member functions (see section 5.3.2), but also by *manipulators*. Manipulators are *inserted* into output streams or extracted from input streams, instead of being activated through the member selection operator (`.'`).

Manipulators are functions. New manipulators can be constructed as well. The construction of manipulators is covered in section 9.9.1. In this section the manipulators that are available in the C++ I/O library are discussed. Most manipulators affect *format flags*. See section 5.3.2 for details about these flags. Most manipulators are parameterless. Sources in which manipulators expecting arguments are used must

```
#include <iomanip>
```

- `ios::boolalpha`:

This manipulator will set the `ios::boolalpha` flag.

- `ios::dec`:

This manipulator enforces the display and reading of integral numbers in decimal format. This is the default conversion. The conversion is applied to values inserted into the stream after processing the manipulators. For example (see also `ios::hex` and `ios::oct`, below):

```

cout << 16 << ", " << hex << 16 << ", " << oct << 16;
// produce the output:
16, 10, 20

```

- `ios::endl:`  
This manipulator will insert a newline character into an output buffer and will flush the buffer thereafter.
- `ios::ends:`  
This manipulator will insert a string termination character into an output buffer.
- `ios::fixed:`  
This manipulator will set the `ios::fixed` flag.
- `ios::flush:`  
This manipulator will flush an output buffer.
- `ios::hex:`  
This manipulator enforces the display and reading of integral numbers in hexadecimal format.
- `ios::internal:`  
This manipulator will set the `ios::internal` flag.
- `ios::left:`  
This manipulator will align values to the left in wide fields.
- `ios::noboolalpha:`  
This manipulator will clear the `ios::boolalpha` flag.
- `ios::noshowpoint:`  
This manipulator will clear the `ios::showpoint` flag.
- `ios::noshowpos:`  
This manipulator will clear the `ios::showpos` flag.
- `ios::noshowbase:`  
This manipulator will clear the `ios::showbase` flag.
- `ios::noskipws:`  
This manipulator will clear the `ios::skipws` flag.
- `ios::nounitbuf:`  
This manipulator will stop flushing an output stream after each write operation. Now the stream is flushed at a `flush`, `endl`, `unitbuf` or when it is closed.
- `ios::nouppercase:`  
This manipulator will clear the `ios::uppercase` flag.
- `ios::oct:`  
This manipulator enforces the display and reading of integral numbers in octal format.

- `ios::resetiosflags(flags):`

This manipulator calls `ios::resetf(flags)` to clear the indicated flag values.

- `ios::right:`

This manipulator will align values to the right in wide fields.

- `ios::scientific:`

This manipulator will set the `ios::scientific` flag.

- `ios::setbase(int b):`

This manipulator can be used to display integral values using the base 8, 10 or 16. It can be used as an alternative to `oct`, `dec`, `hex` in situations where the base of integral values is parameterized.

- `ios::setfill(int ch):`

This manipulator defines the filling character in situations where the values of numbers are too small to fill the width that is used to display these values. By default the blank space is used.

- `ios::setiosflags(flags):`

This manipulator calls `ios::setf(flags)` to set the indicated flag values.

- `ios::setprecision(int width):`

This manipulator will set the precision in which a `float` or `double` is displayed.

- `ios::setw(int width):`

This manipulator expects as its argument the width of the field that is inserted or extracted next. It can be used as manipulator for insertion, where it defines the maximum number of characters that are displayed for the field, but it can also be used during extraction, where it defines the maximum number of characters that are inserted into an array of characters. To prevent array bounds overflow when extracting from `cin`, `setw()` can be used as well:

```
cin >> setw(sizeof(array)) >> array;
```

A nice feature is that a long string appearing at `cin` is split into substrings of at most `sizeof(array) - 1` characters, and that an ASCII-Z character is automatically appended. Notes:

- `setw()` is valid *only* for the next field. It does *not* act like e.g., `hex` which changes the general state of the output stream for displaying numbers.
- When `setw(sizeof(someArray))` is used, make sure that `someArray` really is an array, and not a pointer to an array: the size of a pointer, being, e.g., four bytes, is usually not the size of the array that it points to....

- `ios::showbase:`

This manipulator will set the `ios::showbase` flag.

- `ios::showpoint:`

This manipulator will set the `ios::showpoint` flag.

- `ios::showpos:`

This manipulator will set the `ios::showpos` flag.

- `ios::skipws:`

This manipulator will set the `ios::skipws` flag.

- `ios::unitbuf:`

This manipulator will flush an output stream after each write operation.

- `ios::uppercase:`

This manipulator will set the `ios::uppercase` flag.

- `ios::ws:`

This manipulator will remove all whitespace characters that are available at the current read-position of an input buffer.

## 5.7 The ‘streambuf’ class

The class `streambuf` defines the input and output character sequences that are processed by streams. Like an `ios` object, a `streambuf` object is not directly constructed, but is implied by objects of other classes that are *specializations* of the class `streambuf`.

The class has an important role in realizing possibilities that were available as extensions to the pre-ANSI/ISO standard implementations of C++. Although the class cannot be used directly, its members are introduced here, as the current chapter is the most logical place to introduce the class `streambuf`.

The primary reason for existence of the class `streambuf`, however, is to decouple the `stream` classes from the devices they operate upon. The rationale here is to use an extra software layer between on the one hand the classes allowing us to communicate with the device and the communication between the software 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 for the construction of reusable software, and it is implemented, e.g., in 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 `streambuffer` objects, which in turn communicate with the devices. Thus, as we will see shortly, we are now 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 available to specializing classes only. These *protected members* are listed in this section for further reference. In section 5.7.2 below, a particular specialization of the class `streambuf` is introduced. Note that all public members of `streambuf` discussed here are *also* available in `filebuf`.

In section 14.6 the process of constructing specializations of the class `streambuf` is discussed, and in chapter 19 several other implications of using `streambuf` objects are mentioned. In the current chapter examples of copying streams, of redirecting streams and of reading and writing to streams using the `streambuf` members of `stream` objects are presented (section 5.8).

With the class `streambuf` the following public member functions are available. The type `streamsize` that is used below may, for all practical purposes, be considered an unsigned int.

Public members for input operations:

- `streamsize streambuf::in_avail():`

This member function returns a lower bound on the number of characters that can be read immediately.

- `int streambuf::sbumpc():`

This member function returns the next available character or EOF. The character is removed from the `streambuf` object. If no input is available, `sbumpc()` will call the (protected) member `uflow()` (see section 5.7.1 below) to make new characters available. EOF is returned if no more characters are available.

- `int streambuf::sgetc():`

This member function returns the next available character or EOF. The character is *not* removed from the `streambuf` object, however.

- `int streambuf::sgetn(char *buffer, streamsize n):`

This member function reads `n` characters from the input buffer, and stores them in `buffer`. The actual number of characters read is returned. This member function calls the (protected) member `xsgetn()` (see section 5.7.1 below) to obtain the requested number of characters.

- `int streambuf::snextc():`

This member function removes the current character from the input buffer and returns the next available character or EOF. The character is *not* removed from the `streambuf` object, however.

- `int streambuf::sputback(char c):`

Inserts `c` as the next character to read from the `streambuffer` object. Caution should be exercised when using this function: often at most one character can be put back.

- `int streambuf::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 at most one character can be put back.

Public members for output operations:

- `int streambuf::pubsync():`

Synchronize (i.e., flush) the buffer, by writing any pending information available in the `streambuf`'s buffer to the device. Normally used only by specializing classes.

- `int streambuf::sputc(char c):`

This member function inserts `c` into the `streambuffer` 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 5.7.1 below).

- `int streambuf::sputn(char const *buffer, streamsize n):`

This member function inserts `n` characters from `buffer` into the `streambuffer` object. The actual number of inserted characters is returned. This member function calls the (protected) member `xsputn()` (see section 5.7.1 below) to insert the requested number of characters.

Public members for miscellaneous operations:

- `pos_type streambuf::pubseekoff(off_type offset, ios::seekdir way, ios::openmode mode = ios::in | ios::out):`  
Reset 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. Normally used only by specializing classes.
- `pos_type streambuf::pubseekpos(pos_type offset, ios::openmode mode = ios::in | ios::out):`  
Reset the absolute position of the next character to be read or written to `pos`. Normally used only by specializing classes.
- `streambuf *streambuf::pubsetbuf(char* buffer, streamsize n):`  
Define `buffer` as the buffer to be used by the `streambuf` object. Normally used only by specializing classes.

### 5.7.1 Protected ‘streambuf’ members

The *protected members* of the class `streambuf` are not normally accessible. However, they may be made accessible by specializing classes, and they are important for understanding and using the class `streambuf`. Usually there are both protected data members and protected member functions defined in the class `streambuf`. Since using data members immediately violates the principle of *encapsulation*, these members are not mentioned here. As the functionality of `streambuf`, made available via its member functions, is quite extensive, using its data members directly is probably hardly ever necessary. This section not even lists all protected member functions of the class `streambuf`. Only those member functions are mentioned that are useful in constructing specializations. The class `streambuf` maintains an input- and/or and output buffer, for which beginning, actual and ending pointers are defined, as depicted in figure 5.2. Below this figure is referred to repeatedly.

Protected constructor:

- `streambuf::streambuf():`  
Default (protected) constructor of the class `streambuf`.

Protected member functions related to input operations:

- `char *streambuf::eback():`  
For the input buffer the class `streambuf` maintains three pointers: `eback()` points to the ‘end of the putback’ area: characters can safely be put back up to this position. See also figure 5.2. `Eback()` can be considered to represent the *beginning* of the input buffer.
- `char *streambuf::egptr():`  
For the input buffer the class `streambuf` maintains three pointers: `egptr()` points just beyond the last character that can be retrieved. See also figure 5.2. If `gptr()` (see below) equals `egptr()` the buffer must be refilled. This is realized by calling `underflow()`, see below.



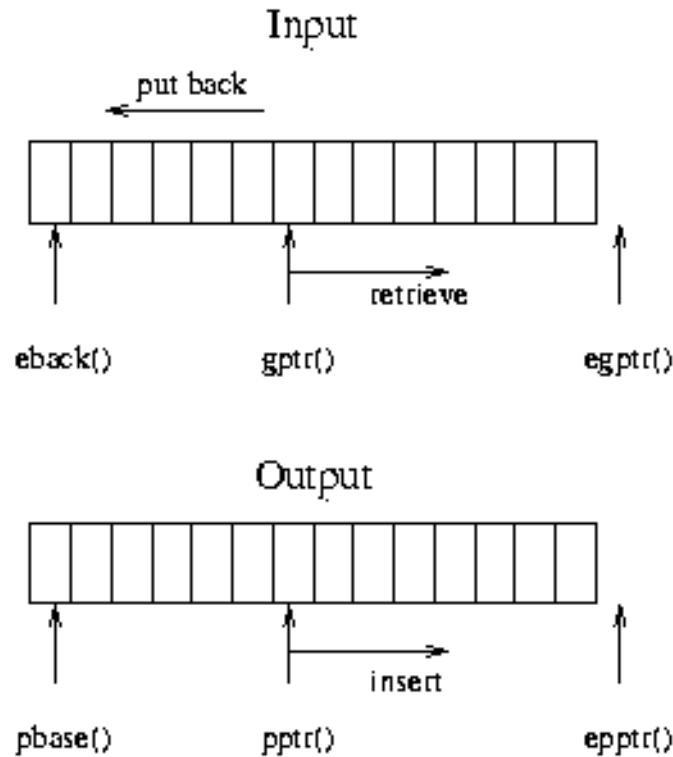


Figure 5.2: Input- and output buffer pointers of the class 'streambuf'

- `void streambuf::gbump(int n):`

This function moves the input pointer over `n` positions.

- `char *streambuf::gptr():`

For the input buffer the class `streambuf` maintains three pointers: `gptr()` points to the next character to be retrieved. See also figure 5.2.

- `int streambuf::pbackfail(int c):`

This member function may be redefined by specializations of the class `streambuf` to do something intelligent when putting back character `c` fails. One of the things to consider here is to restore the old read pointer when putting back a character fails, because the beginning of the input buffer is reached. This member function is called when ungetting or putting back a character fails.

- `void streambuf::setg(char *beg, char *next, char *beyond):`

This member function 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 beyond the last character of the input buffer. Usually `next` is at least `beg + 1`, to allow for a put back operation. No input buffering is used when this member is called with 0-arguments.

- `streamsize streambuf::showmanyc():`

(Pronounce: s-how-many-c) This member function may be redefined by specializations of the class `streambuf`. 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 at least 0 characters will be returned before the latter two functions will return EOF).

- `int streambuf::uflow():`

This member function may be redefined by specializations of the class `streambuf` to reload an input buffer with new characters. The default implementation is to call `underflow()`, see below, and to increment the read pointer `gptr()`.

- `int streambuf::underflow():`

This member function may be redefined by specializations of the class `streambuf` to read another character from the device. The default implementation is to return EOF. When buffering is used, often the complete buffer is not refreshed, as this would make it impossible to put back characters just after a reload. This system, where only a subsection of the input buffer is reloaded, is called a *split buffer*.

- `streamsize streambuf::xsgetn(char *buffer, streamsize n):`

This member function may be redefined by specializations of the class `streambuf` to retrieve `n` characters from the device. The default implementation is to call `sbumpc()` for every single character. By default this calls (eventually) `underflow()` for every single character.

Protected member functions related to output operations:

- `int streambuf::overflow(int c):`

This member function may be redefined by specializations of the class `streambuf` to flush the characters in the output buffer to the device, and then to reset the output buffer pointers such that the buffer may be considered empty. If no output buffering is used, `overflow()` is called for every single character which is written to the `streambuf` object. This is realized by setting the buffer pointers (using, e.g., `setp()`, see below) to 0. The default implementation returns EOF, indicating that no characters can be written to the device.

- `char *streambuf::pbase():`

For the output buffer the class `streambuf` maintains three pointers: `pbase()` points to the beginning of the output buffer area. See also figure 5.2.

- `char *streambuf::eptr():`

For the output buffer the class `streambuf` maintains three pointers: `eptr()` points just beyond the location of the last character that can be written. See also figure 5.2. If `pptr()` (see below) equals `eptr()` the buffer must be flushed. This is realized by calling `overflow()`, see below.

- `void streambuf::pbump(int n):`

This function moves the output pointer over `n` positions.

- `char *streambuf::pptr():`

For the output buffer the class `streambuf` maintains three pointers: `pptr()` points to the location of the next character to be written. See also figure 5.2.

- `void streambuf::setp(char *beg, char *beyond):`

This member function initializes an output buffer: `beg` points to the beginning of the output area and `beyond` points beyond the last character of the output area. Use 0 for the arguments to indicate that no buffering is requested. In that case `overflow()` is called for every single character to write to the device.

- `streamsize streambuf::xsputn(char const *buffer, streamsize n):`

This member function may be redefined by specializations of the class `streambuf` to write `n` characters immediately to the device. The default implementation calls `sputc()` for each individual character, so redefining is only needed if a more efficient implementation is required.

Protected member functions related to buffer management and positioning:

- `streambuf *streambuf::setbuf(char *buffer, streamsize n):`

This member function may be redefined by specializations of the class `streambuf` to install a buffer. The default implementation is to do nothing.

- `pos_type streambuf::seekoff(off_type offset, ios::seekdir way, ios::openmode mode = ios::in | ios::out):`

This member function may be redefined by specializations of the class `streambuf` to reset the next pointer for input or output to a new relative position. The default implementation is to indicate failure by returning -1.

- `pos_type streambuf::seekpos(pos_type offset, ios::openmode mode = ios::in | ios::out):`

This member function may be redefined by specializations of the class `streambuf` to reset the next pointer for input or output to a new absolute position. The default implementation is to indicate failure by returning -1.

- `int sync():`

This member function may be redefined by specializations of the class `streambuf` to flush the output buffer to the device or to reset the input device to the position of the last consumed character. The default implementation (not using a buffer) is to return 0, indicating successful syncing. The member function is used to make sure that any characters that are still buffered are written to the device or to restore unconsumed characters to the device when the `streambuf` object ceases to exist.

Morale: when specializations of the class `streambuf` are designed, the very least thing to do is to redefine `underflow()` for specializations aimed at reading information from devices, and to redefine `overflow()` for specializations aimed at writing information to devices. Several examples of specializations of the class `streambuf` will be given in the sequel.

### 5.7.2 The class 'filebuf'

The class `filebuf` is a specialization of `streambuf` used by the file stream classes. Apart from the (public) members that are available through the class `streambuf`, it defines the following extra (public) members:

- `filebuf::filebuf()`:

Since the class has a constructor, it is, different from the class `streambuf`, possible to construct a `filebuf` object. This defines a plain `filebuf` object, not yet connected to a stream.

- `bool filebuf::is_open()`:

This member function returns `true` if the `filebuf` is actually connected to an open file. See the `open()` member, below.

- `filebuf *filebuf::open(char const *name, ios::openmode mode)`:

This member function associates the `filebuf` object with a file whose name is provided. The file is opened according to the provided `ios::openmode`.

- `filebuf *filebuf::close()`:

This member function closes the association between the `filebuf` object and its file. The association is automatically closed when the `filebuf` object ceases to exist.

## 5.8 Advanced topics

### 5.8.1 Copying streams

Usually, files are copied either by reading a source file character by character or line by line. The basic *mold* for processing files is as follows:

- In an eternal loop:
  1. read a character
  2. if reading did not succeed (i.e., `fail()` returns true), break from the loop
  3. process the character

It is important to note that the reading must *precede* the testing, as it is only possible to know after the actual attempt to read from a file whether the reading succeeded or not. Of course, variations are possible: `getline(istream &, string &)` (see section 5.5.1) returns an `istream &` itself, so here reading and testing may be realized in one expression. Nevertheless, the above mold represents the general case. So, the following program could be used to copy `cin` to `cout`:

```
#include <iostream>

using namespace::std;

int main()
{
    while (true)
    {
        char c;

        cin.get(c);
        if (cin.fail())
            break;
    }
}
```

```

        cout << c;
    }
    return 0;
}

```

By combining the `get()` with the `if`-statement a construction comparable to `getline()` could be used:

```

    if (!cin.get(c))
        break;

```

Note, however, that this would still follow the basic rule: ‘read first, test later’.

This simple copying of a file, however, isn't required very often. More often, a situation is encountered where a file is processed up to a certain point, whereafter the remainder of the file can be copied unaltered. The following program illustrates this situation: the `ignore()` call is used 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 used a special overloaded version of the `<<`-operator, in which a `streambuf` pointer is inserted into another stream. As the member `rdbuf()` returns a `streambuf *`, it can thereupon be inserted into `cout`. This immediately copies the remainder of `cin` to `cout`:

```

#include <iostream>
using namespace std;

int main()
{
    cin.ignore(80, '\n');    // skip the first line
    cout << cin.rdbuf();     // copy the rest by inserting a streambuf *
}

```

Note that this method assumes a `streambuf` object, so it will work for all specializations of `streambuf`. Consequently, if the class `streambuf` is specialized for a particular device it can be inserted into any other stream using the above method.

## 5.8.2 Coupling streams

Ostreams can be *coupled* to `ios` objects using the `tie()` member function. This results in flushing any buffered output of the `ostream` object (by calling `flush()`) when any input or output is performed on the `ios` object to which the `ostream` object is tied. By default `cout` is tied to `cout`. To break coupling, the member function `ios::tie(0)` can be called.

Another (frequently useful) example of coupling streams is to tie `cerr` to `cout`: this way standard output and error messages written to the screen will appear in sync with the time at which they were generated:

```

#include <iostream>
using namespace std;

int main()
{
    cout << "first (buffered) line to cout\n";
    cerr << "first (unbuffered) line to cerr\n";
}

```

```

    cerr.tie(&cout);

    cout << "second (buffered) line to cout\n";
    cerr << "second (unbuffered) line to cerr\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 realized 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 `iostream` library rather than operating system calls can be realized. See the next sections for examples.

### 5.8.3 Redirection using streams

By using the `ios::rdbuf()` member streams can share their `streambuf` objects. This means that the information that is written to a stream will actually be written to another stream, a phenomenon normally called *redirection*. Redirection is normally realized at the level of the operating system, and in some situations that is still necessary (see section 19.3).

A standard situation where redirection is wanted is to write error messages to file rather than to the standard error file, 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>/tmp/error.log
```

With this command any error messages written by `program` will be saved on the file `/tmp/error.log`, rather than being written to the screen.

Here is how this can be realized using `streambuf` objects. Assume `program` now expects an optional argument defining the name of the file to write the error messages to; so `program` is now called as:

```
program /tmp/error.log
```

Here is the example realizing redirection. It is annotated below.

```

#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char **argv)
{
    ofstream                                     // 8
        errlog;

```

```

    streambuf
        *cerr_buffer = 0;                                // 11

    if (argc == 2)
    {
        errlog.open(argv[1]);                             // 15
        cerr_buffer = cerr.rdbuf(errlog.rdbuf());         // 16
    }
    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();                                            // 29

    cerr << "Several messages to stderr, msg 3\n";

    cerr.rdbuf(cerr_buffer);                             // 33
    cerr << "Done\n";                                    // 34
}
/*
    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 8-11 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. This is further discussed below.
- At line 15 the alternate error stream is opened.
- At line 16 the redirection takes place: `cerr` will now write to the `streambuf` defined by `errlog`. It is important that the original buffer used by `cerr` is saved, as explained below.
- At line 29 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 33 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 34, Done is now written to the screen again, as the redirection has been terminated.

#### 5.8.4 Reading AND Writing to a stream

In order to be able to read and write to a stream an `fstream` object must be created. As with `ifstream` and `ofstream` objects, the constructor receives the name of the file to be opened:

```
fstream inout("iofile", ios::in | ios::out);
```

Note the use of the `ios` 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 binary or operator `|`. Alternatively, instead of `ios::out`, `ios::app` could have been used, in which case writing will always be done at the end of the file.

Somehow reading and writing to a file is a bit awkward: what to do when the file may or may not exist yet, but if it already exists it should not be rewritten? I have been fighting with this problem for some time, and now use the following approach:

```
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    fstream rw("fname", ios::out | ios::ate | ios::in);
    if (!rw)
    {
        rw.clear();
        rw.open("fname", ios::out | ios::trunc | ios::in);
    }
    if (!rw)
    {
        cerr << "Opening 'fname' failed miserably" << endl;
        return 1;
    }

    cerr << rw.tellp() << endl;

    rw << "Hello world" << endl;
    rw.seekg(0);

    string s;
    getline(rw, s);

    cout << "Read: " << s << endl;
}
```



In the above example, the constructor fails when `fname` doesn't exist yet. However, in that case the `open()` member succeeds. If the file already exists, the constructor succeeds. Because of the `ios::ate` flag, the next read/write action will by default take place at EOF. However, `ios::ate` is not `ios::app`, and the stream object can be repositioned using `seekg()` or `seekp()`.

Under **DOS**-like operating systems, which use the multiple character `\r\n` sentinels to separate lines in text files the flag `ios::bin` is required for processing binary files to ensure that `\r\n` combinations are processed as two characters.

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 5.4.2 for details.

Once a file has been opened in read and write mode, the `<<` operator can be used to insert information to the file, while the `>>` operator may be used to extract information from the file. These operations may be performed in random order. The following fragment will read a blank-delimited word from the file, and will then write a string to the file, just beyond the point where the string just read terminated, followed by the reading of yet another string just beyond the location where the string just written ended:

```
fstream
    f("filename", ios::in | ios::out | ios::creat);
string
    str;

f >> str;          // read the first word
                   // write a well known text
f << "hello world";
f >> str;          // and read again
```

Since the operators `<<` and `>>` can apparently be used with `fstream` objects, you might wonder whether a series of `<<` and `>>` operators in one statement might be possible. After all, `f >> str` should produce a `fstream &`, shouldn't it?

The answer is: it doesn't. The compiler casts the `fstream` object into an `istream` object in combination with the extraction operator, and into an `ostream` object in combination with the insertion operator. Consequently, a statement like

```
f >> str << "grandpa" >> str;
```

results in a compiler error like

```
no match for 'operator <<(class istream, char[8])'
```

Since the compiler complains about the `istream` class, the `fstream` object is apparently considered an `istream` object in combination with the extraction operator.

Of course, random insertions and extractions are hardly used. Generally, insertions and extractions take place at specific locations in the file. In those cases, the position where the insertion or extraction must take place can be controlled and monitored by the `seekg()` and `tellg()` member functions (see sections 5.4.1 and 5.5.1).

Error conditions (see section 5.3.1) occurring due to, e.g., reading beyond end of file, reaching end of file, or positioning before begin of file, can be cleared using the `clear()` member function. Following `clear()` processing may continue. E.g.,

```

fstream
    f("filename", ios::in | ios::out | ios::creat);
string
    str;

f.seekg(-10); // this fails, but...
f.clear();    // processing f continues

f >> str;    // read the first word

```

A common situation in which files are both read and written occurs in *data base* applications, where files consists of records of fixed size, or where the location and size of pieces of information is well known. For example, the following program may be used to add lines of text to a (possibly existing) file, and to retrieve a certain line, based on its order-number from the file. Note the use of the *binary file* index to retrieve the location of the first byte of a line.

```

#include <iostream>
#include <fstream>
#include <string>

using namespace std;

void err(char const *msg)
{
    cout << msg << endl;
    return;
}

void err(char const *msg, long value)
{
    cout << msg << value << endl;
    return;
}

void read(fstream &index, fstream &strings)
{
    int
        idx;

    if (!(cin >> idx)) // read index
        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 << endl;      // 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)
        )
    )
        err("Writing failed to index: ", offset);

    if (!(strings << line << endl))             // write the line itself
        err("Writing to 'strings' failed");

    // confirm writing the line
    cout << "Write at offset " << offset << " line: " << line << endl;
}

int main()
{
    fstream
        index("index", ios::trunc | ios::in | ios::out),
        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)
{
    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, strings);
    else if (cmd == "w")
        write(index, strings);
    else
        cout << "Unknown command: " << cmd << endl;
}
}

```

As another example of reading and writing files, consider the following program, which also serves as an illustration of reading an ASCII-Z delimited string:

```

#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    fstream                                // r/w the file
        f("hello", ios::in | ios::out | ios::trunc);

    f.write("hello", 6);                    // write 2 ascii-z
    f.write("hello", 6);

    f.seekg(0, ios::beg);                    // reset to begin of file

    char
        buffer[20],
        c;

                                // read the first 'hello'
    cout << f.get(buffer, 100, 0).tellg() << endl;;
    f >> c;                                // read the ascii-z delim

                                // and read the second 'hello'
    cout << f.get(buffer + 6, 100, 0).tellg() << endl;

    buffer[5] = ' ';                        // change asciiz to ' '
    cout << buffer << endl;                // show 2 times 'hello'
}
/*
Generated output:

```

```

5
11
hello hello
*/

```

By using the `streambuf` members of stream objects a completely different way to both read and write to streams can be implemented. All mentioned considerations remain valid: before a read operation following a write operation a `seekg()` must be used, and before a write operation following a read operation a `seekp()` must be used. When the stream's `streambuf` objects are used, either an `istream` is associated with the `streambuf` object of another `ostream` object, or *vice versa*, an `ostream` object is associated with the `streambuf` object of another `istream` object. Here is the same program as before, now using *associated streams*:

```

#include <iostream>
#include <fstream>
#include <string>

using namespace std;

void err(char const *msg)
{
    cout << msg << endl;
    return;
}

void err(char const *msg, long value)
{
    cout << msg << value << endl;
    return;
}

void read(istream &index, istream &strings)
{
    int
        idx;

    if (!(cin >> idx))                // read index
        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 << endl;      // show the line
}

void write(ostream &index, ostream &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)
        )
    )
        err("Writing failed to index: ", offset);

    if (!(strings << line << endl))             // write the line itself
        err("Writing to 'strings' failed");

    // confirm writing the line
    cout << "Write at offset " << offset << " line: " << line << endl;
}

int main()
{
    ifstream
        index_in("index", ios::trunc | ios::in | ios::out),
        strings_in("strings", ios::trunc | ios::in | ios::out);
    ostream
        index_out(index_in.rdbuf()),
        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 << endl;
}
}

```

Please note:

- The streams to be associated with the `streambuf` objects of existing streams are not `ifstream` or `ofstream` objects (or, for that matter, `istreamstream` or `ostreamstream` objects), but basic `istream` and `ostream` objects.
- The `streambuf` object does not have to be defined in an `ifstream` or `ofstream` object: it can be defined outside of the streams, using constructions like:

```

filebuf
    fb("index", ios::in | ios::out | ios::trunc);
istream
    index_in(&fb);
ostream
    index_out(&fb);

```

- Note that an `ifstream` object can be constructed using stream modes normally used for writing to files. Conversely, `ofstream` objects can be constructed using stream modes normally used for reading from files.
- If `istream` and `ostreams` are associated through a common `streambuf`, then the read and write pointers (should) point to the same locations: they are tightly coupled.
- The advantage of using a 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 then specifically be constructed to interface particular devices. Elaborating this is left as an exercise to the reader.

## Chapter 6

# Classes

In this chapter classes are the topic of discussion. Two special member functions, the constructor and the destructor, are introduced.

In steps we will construct a class `Person`, which could be used in a database application to store a name, an address and a phone number of a person.

Let's start off by introducing the declaration of a class `Person` right away. The class declaration is normally contained in the *header file* of the class, e.g., `person.h`. The class declaration is generally not called a *declaration*, though. Rather, the common name for class declarations is *class interface*, to be distinguished from the definitions of the function members, called the *class implementation*. Thus, the interface of the class `Person` is given next:

```
#include <string>

class Person
{
    public:                // interface functions
        void setname(string const &n);
        void setaddress(string const &a);
        void setphone(string const &p);
        void setweight(unsigned weight);

        string const &getname(void) const;
        string const &getaddress(void) const;
        string const &getphone(void) const;
        unsigned getweight() const;

    private:              // data fields
        string
            name,          // name of person
            address,       // address field
            phone;          // telephone number
        unsigned
            weight;         // the weight in kg.
};
```

The data fields in this class are `name`, `address`, `phone` and `weight`. All fields (but `weight`) are `string` objects. The data are `private`, which means that they can only be accessed by the functions of the



```
class Person.
```

The data are manipulated by interface functions which take care of all communication with code outside of the class. Either to set the data fields to a given value (e.g., `setname()`) or to inspect the data (e.g., `getname()`).

Note once again how similar the `class` is to the `struct`. The fundamental difference being that by default classes have *private* members, whereas structs have *public* members. Since the convention calls for the public members of a class to appear first, the keyword `private` is needed to switch back from public members to the (default) private situation.

## 6.1 The constructor

A class in C++ may contain two special categories of member functions which are involved in the internal workings of the class. These member function categories are, on the one hand, the constructors and, on the other hand, the destructor. The primary function of the *destructor* is returning memory allocated by an object when an object goes 'out of scope'. Allocation of memory is discussed in chapter 7, and the discussion of destructors is therefore postponed until that chapter.

This chapter, however, will concentrate on the basic form of the `class` and of their constructor members.

The constructor member function has by definition the same name as the corresponding class. The constructor does not specify a return value, not even `void`. E.g., for the class `Person` the constructor is `Person::Person()`. The C++ run-time system makes sure that the constructor of a class, if defined, is called when a variable of the class, called an object, is created. It is of course possible to define a class which has no constructor at all. In that case the run-time system either calls no function or it calls a dummy constructor (i.e., a constructor which performs no actions) when a corresponding object is created. The actual generated code of course depends on the compiler<sup>1</sup>.

Objects may be defined at a local (function) level, or at a global level (in which its status is comparable to a global variable).

When an object is locally defined (in a function), the constructor is called every time the function is called. The object's constructor is then called at the point where the variable is defined (a subtlety here is that a variable may be defined implicitly as, e.g., a temporary variable in an expression).

When an object is defined as a static object (i.e., it is static variable) in a function, the constructor is called when the function in which the static variable is defined is called for the first time.

When an object is defined as a global object the constructor is called when the program starts. Note that in this case the constructor is called even before the function `main()` is started. This feature is illustrated in the following program:

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

class Demo
{
public:
    Demo();
};
```

---

<sup>1</sup>A compiler-supplied constructor in a class which contains composed objects (see section 6.4) will 'automatically' call the member initializers, and therefore does perform some actions. We postpone the discussion of such constructors to 6.4.1.

```

Demo::Demo()
{
    cout << "Demo constructor called\n";
}

Demo
    d;

int main()
{}

/*
    Generated output:
    Demo constructor called
*/

```

The above listing shows how a class `Demo` is defined which consists of just one function: the constructor. The constructor performs but one action: a message is printed. The program contains one global object of the class `Demo`, and `main()` has an empty body. Nonetheless, the program produces some output.

Some important characteristics of constructors are:

- The constructor has the same name as its class.
- The primary function of a constructor is to make sure that all its data members have sensible or at least defined values once the object has been constructed. We'll get back to this important task shortly.
- The constructor does not have a return value. This is true for the declaration of the constructor in the class definition, as in:

```

class Demo
{
    public:
        Demo();          // no return value here
};

```

and it also holds true for the definition of the constructor function, as in:

```

Demo::Demo()            // no return value here
{
    // statements ...
}

```

- The constructor function in the example above has no arguments. It is called the *default constructor*. That a constructor has no arguments is, however, no requirement *per se*. We shall shortly see that it is possible to define constructors *with* arguments as well as *without* arguments.

**NOTE:** Once a constructor is defined having arguments, the default constructor doesn't exist anymore, unless the default constructor is defined explicitly too.

The above note has important consequences, as the default constructor is required in cases where it must be able to construct an object either *with* or *without* explicit initialization values.

By merely defining a constructor having at least one arguments, the implicitly available default constructor disappears from view. As noted, to make it available again in this situation, it must be defined explicitly too.

### 6.1.1 A first application

As illustrated at the beginning of this chapter, the class `Person` contains three `private string` data members and an `unsigned weight` data member. These data members can be manipulated by the interface functions. The internal workings of the class are as follows: when a name, address or phone number or weight of a `Person` is defined, the corresponding private data member is given a (new) value. An obvious setup is as follows:

- The assignment to a data member (using a `set...()` function) consists of the assignment of the new value to the corresponding data member.
- Inspecting a data member by means of one of the `get...()` functions simply returns the corresponding data member

The `set...()` functions could be constructed as follows:

```
#include "person.h"                // given earlier

// interface functions set...()
void Person::setname(string const &n)
{
    name = n;
}

void Person::setaddress(string const &a)
{
    address = a;
}

void Person::setphone(string const &p)
{
    phone = p;
}

void Person::setweight(unsigned w)
{
    weight = w;
}
```

The `get...()` interface functions are now defined. Note the occurrence of the keyword `const` following the parameter lists of the functions: the member functions are *const member functions*, indicating that they will not modify their *object* when they're called. Furthermore, notice that the return values of the member functions of the `string` data members are `string const &` values: the `const` here indicates that the *caller* of the memberfunction *cannot* alter the returned value itself. The caller of the `get...()` member function *could* copy the returned value to a variable of its own, though, and *that* variable's value may then of course be modified *ad lib*. Const member functions are discussed in more detail in section 6.2. The return value of the `getweight()` member function,

however, is a plain unsigned, as this can be a simple copy of the value that's stored in the `Person`'s `weight` member:

```
#include "person.h"                // given earlier

// interface functions get...()
string const &Person::getname() const
{
    return name;
}

string const &Person::getaddress() const
{
    return address;
}

string const &Person::getphone() const
{
    return phone;
}

unsigned Person::getweight() const
{
    return weight;
}
```

The class definition of the `Person` class given earlier can still be used. The `set...()` and `get...()` functions merely implement the member functions declared in that class definition.

To demonstrate the use of the class `Person`, an example is now given. An object is initialized and passed to a function `printperson()`, which prints the person's data. Note also the usage of the reference operator `&` in the argument list of the function `printperson()`. This way only a reference to an existing `Person` object is passed, rather than a whole object. The fact that `printperson()` does not modify its argument is evident from the fact that the argument is declared `const`.

Alternatively, the function `printperson()` could have been defined as a public member function of the class `Person`, rather than a plain, objectless function.

```
#include <iostream>
#include "person.h"                // given earlier

void printperson(Person const &p)
{
    cout << "Name      : " << p.getname()      << endl <<
         "Address   : " << p.getaddress()    << endl <<
         "Phone     : " << p.getphone()      << endl <<
         "Weight    : " << p.getweight()     << endl;
}

int main()
{
    Person
        p;
```

```

        p.setname("Linus Torvalds");
        p.setaddress("E-mail: Torvalds@cs.helsinki.fi");
        p.setphone(" - not sure - ");
        p.setweight(75);           // kg.

        printperson(p);
        return 0;
    }
/*
    Produced output:

Name      : Linus Torvalds
Address   : E-mail: Torvalds@cs.helsinki.fi
Phone     : - not sure -
Weight    : 75
*/

```

### 6.1.2 Constructors: with and without arguments

In the above declaration of the class `Person` the constructor has no arguments. C++ allows constructors to be defined with or without argument lists. The arguments are supplied when an object is created.

For the class `Person` a constructor may be handy which expects three strings: the name, address and phone number. Such a constructor is, for example:

```

Person::Person(string const &n, string const &a, string const &p,
               unsigned w)
{
    name = n;
    address = a;
    phone = p;
    weight = w;
}

```

The constructor must be declared in the class declaration as well:

```

class Person
{
    public:
        Person::Person(string const &name, string const &address,
                       string const &phone, unsigned weight);

        // rest of the class interface
};

```

However, now that this constructor has been declared, the default constructor must be declared explicitly too, if we still want to be able to construct a plain `Person` object without any specific value of its name, address and phone data members.

Since C++ allows function overloading, such a declaration of a constructor can co-exist with a constructor without arguments. The class `Person` would thus have two constructors, and the relevant part of the class interface becomes:

```

class Person
{
    public:
        Person();
        Person(string const &name, string const &address,
              string const &phone, unsigned weight);

        // rest of the class interface
};

```

In this case, the `Person()` constructor doesn't have to do much, as it doesn't have to initialize the `string` data members of the `Person` object: these data members are by default initialized to empty strings themselves. However, there is also an `unsigned` data member. That member is a variable of a basic type, contrary to the `string` members, which already use constructors of their own. So, unless the value of the `weight` data member is explicitly initialized, it will be

- 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, the default constructor has a job to do: initializing the data members which are not initialized to sensible values automatically. Here is an implementation of the default constructor:

```

Person::Person()
{
    weight = 0;
}

```

The use of a constructor with and without arguments (i.e., the default constructor) is illustrated in the following code fragment. The object `a` is initialized at its definition using the constructor with arguments, with the `b` object the default constructor is used:

```

int main()
{
    Person
        a("Karel", "Rietveldlaan 37", "542 6044", 70),
        b;

    return 0;
}

```

In this example, the `Person` objects `a` and `b` are created when `main()` is started: they are *local* objects, living for as long as the `main()` function is active.

If `Person` objects must be constructed using other arguments, other constructors are required as well. It is also possible to define default parameter values. These default parameter values must be given in the class interface, e.g.,

```

class Person
{
    public:
        Person();

```

```

        Person::Person(string const &name,
                        string const &address = "--unknown--",
                        string const &phone   = "--unknown--",
                        unsigned weight = 0);

        // rest of the class interface
};

```

## The order of construction

The possibility to pass arguments to constructors offers us the chance to monitor at which moment in a program's execution an object is created. This is shown in the next listing, using a class `Test`. The program listing below shows a class `Test`, a global `Test` object, and two local `Test` objects: in a function `func()` and in the `main()` function. 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:

```

#include <iostream>
#include <string>

class Test
{
public:
    Test(string const &name);    // constructor with an argument
};

Test::Test(string const &name)
{
    cout << "Test object " << name << " created" << endl;
}

Test
    globaltest("global");

void func()
{
    Test
        functest("func");
}

int main()
{
    Test
        first("main first");
    func();
    Test
        second("main second");
    return 0;
}

/*
    Generated output:
Test object global created
Test object main first created

```

```

Test object func created
Test object main second created
*/

```

## 6.2 Const member functions and const objects

The keyword `const` is often seen in the declarations of member functions following the argument list. This keyword is used to indicate that a member function does not alter the data fields of its object, but only inspects them. These member functions are called *const member functions*. Using the example of the class `Person`, we see that the `get...()` functions were declared `const`:

```

class Person
{
    public:
        string const &getname(void) const;
        string const &getaddress(void) const;
        string const &getphone(void) const;
};

```

As illustrated in this fragment, the keyword `const` appears *following* the argument list of functions. Note that in this situation the rule of thumb given in section 3.1.3 applies once again: whichever appears **before** the keyword `const`, may not be altered and doesn't alter (its own) data.

The `const` specification must be repeated in the definition of member functions themselves:

```

string const &Person::getname() const
{
    return name;
}

```

A member function which is declared and defined as `const` may not alter any data fields of its class. In other words, a statement like

```

name = 0;

```

in the above `const` function `getname()` would result in a compilation error.

`Const` member functions exist because C++ allows `const` objects to be created, or references to `const` objects to be passed to functions. For such objects only member functions which do not modify it, i.e., the `const` member functions, may be called. The only exception to this rule are the constructors and destructor: these are called 'automatically'. The possibility of calling constructors or destructors is comparable to the definition of a variable `int const max = 10`. In situations like these, no *assignment* but rather an *initialization* takes place at creation-time. Analogously, the constructor can **initialize** its object when the `const` variable is created, but subsequent assignments cannot take place.

The following example shows how a `const` object of the class `Person` can be defined. When the object is created the data fields are initialized by the constructor:

```

Person const
    me("Karel", "karel@icce.rug.nl", "542 6044");

```



Following this definition it would be illegal to try to redefine the name, address or phone number for the object `me`: a statement as

```
me.setname("Lerak");
```

would not be accepted by the compiler. Once more, look at the position of the `const` keyword in the variable definition: `const`, following `Person` and preceding `me` associates to the left: the `Person` object in general must remain unaltered. Hence, if multiple objects were defined here, both would be constant `Person` objects, as in:

```
Person const          // all constant Person objects
    kk("Karel", "karel@icce.rug.nl", "542 6044"),
    fbb("Frank", "f.b.brokken@rc.rug.nl", "363 3688");
```

Member functions which do not modify their object should be defined as `const` member functions. This subsequently allows the use of these functions with `const` objects or with `const` references.

## 6.3 The keyword ‘inline’

Let us take another look at the implementation of the function `Person::getname()`:

```
string const &Person::getname() const
{
    return name;
}
```

This function is used to retrieve the name field of an object of the class `Person`. In a code fragment like:

```
Person
    frank("Frank", "Oostumerweg 17", "403 2223");

cout << frank.getname();
```

the following actions take place:

- The function `Person::getname()` is called.
- This function returns the name of the object `frank` as a reference.
- The referenced name is inserted into `cout`.

Especially the first part of these actions results in some time loss, since an extra function call is necessary to retrieve the value of the `name` field. Sometimes a faster procedure may be desirable, in which the `name` field becomes immediately available; thus avoiding the call to `getname()`. This can be realized by using `inline` functions, which can be defined in two ways.

### 6.3.1 Inline functions within class declarations

Using the first method to implement inline functions, the code of a function is defined *in a class declaration itself*. For the class `Person` this would lead to the following implementation of `getname()`:

```
class Person
{
    public:
        string const &getname(void) const
        {
            return name;
        }
};
```

Note that the inline code of the function `getname()` now literally occurs inline in the interface of the class `Person`. The keyword `const` occurs after the function declaration, and before the code block.

Thus, inline functions appearing in the class interface show their full (and standard) definition *within* the class interface itself.

The effect of this is the following. When `getname()` is called in a program statement, the compiler *generates the code of the function* when the function is used in the source text, rather than a call to the function, appearing only once in the compiled program.

This construction, where the function code itself is inserted rather than a call to the function, is called an inline function. Note that the use of inline function results in duplication of the code of the function 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 considers the use of inline functions a *request* rather than a *command*: if the compiler considers the function too long, it will not grant the request, but will, instead, treat the function as a normal function.

### 6.3.2 Inline functions outside of class declarations

The second way to implement inline functions leaves a class interface intact, but mentions the keyword `inline` in the function definition. The interface and implementation in this case are as follows:

```
class Person
{
    public:
        string const &getname(void) const;
};

inline string const &Person::getname() const
{
    return name;
}
```

Again, the compiler will insert the code of the function `getname()` instead of generating a call.

The `inline` function must still appear in the same file as the class interface, and cannot be compiled to be stored in, e.g., a library. The reason for this is that the *compiler* rather than the *linker* must be able to insert the code of the function in a source text offered for compilation. Code stored in a library is inaccessible to the compiler. Consequently, inline functions are always defined together with the class interface.

### 6.3.3 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?
- Defining inline functions can be considered once a fully developed and tested program runs too slowly and shows 'bottlenecks' in certain functions. A profiler, which runs a program and determines where most of the time is spent, is necessary for such optimization.
- inline functions can be used when member functions consist of one very simple statement (such as the return statement in the function `Person::getname()`).
- By defining a function as `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.
- It is only useful to implement an inline function when the time which is spent during a function call is long compared to the code in the function. An example where an inline function has hardly any effect at all is the following:

```
void Person::printname() const
{
    cout << name << endl;
}
```

This function, which is, for the sake of the example, presented as a member of the class `Person`, 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 a very insignificant gain in execution time.

All inline functions have one disadvantage: the actual code is inserted by the compiler and must therefore be known 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 placed 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 blurring the distinction between interface and implementation.

Finally, note once again that using the keyword `inline` is not really a *command* for the compiler. Rather, it is a *request* the compiler may or may not grant.

## 6.4 Objects in objects: composition

Often objects are used as data members in class definitions. This is referred to as *composition*.

For example, the class `Person` holds information about the name, address and phone number. This information is stored in `string` data members, which are themselves objects: composition.

Composition is not extraordinary or C++ specific: in C it is quite common to include a `struct` or `union` in other compound types.

The initialization of composed objects deserves some extra attention: the topics of the coming sections.

### 6.4.1 Composition and const objects: const member initializers

Composition of objects has an important consequence for the constructor functions of the 'composed' (embedded) object. Unless explicitly instructed otherwise, the compiler generates code to call the default constructors of all composed classes in the constructor of the composing class.

Often it is desirable to initialize a composed object from a specific constructor of the composing class. This is illustrated below for the class `Person`. In this fragment it is assumed that a constructor for a `Person` should be defined expecting four arguments: the name, address and phone number plus the person's weight:

```
Person::Person(char const *nm, char const *adr,
               char const *ph, unsigned weight)
:
    name(nm),
    address(adr),
    phone(ph),
    weight(weight)
{ }
```

Following the argument list of the constructor `Person::Person()`, the constructors of the `string` data members are explicitly called, e.g., `name(mn)`. The initialization takes place **before** the code block of `Person::Person()` (now empty) is executed. This construction, where member initialization takes place before the code block itself is executed is called *member initialization*. Member initialization can be made explicit in the *member initializer list*, that may appear after the parameter list, between a colon (announcing the start of the member initializer list) and the opening curly brace of the code block of the constructor.

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* automatically initialized.

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 `weight` from the parameter `weight`. For further illustration, we used the same identifiers here: with member initialization there is no ambiguity and the first (left) identifier in `weight(weight)` is always interpreted as the data member to be initialized, whereas the identifier between parentheses is interpreted as the parameter.

When a class has multiple composed data members, all members can be initialized using a 'member initializer list': this list consists of the constructors of all composed objects, separated by commas.

The *order* in which the objects are initialized is defined by the order in which the members are defined in the 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 much as possible: it can be downright necessary to use them, and *not* using member initializers can result in inefficient code: with objects always at least the default constructor is called. So, in the following example, first the `string` members are initialized to empty strings, whereafter these values are immediately redefined to their intended values. Of course, the immediate initialization to the intended values would have been more efficient.

```
Person::Person(char const *nm, char const *adr,
               char const *ph, unsigned wt)
{
    name = nm;
    address = adr;
    phone = ph;
    weight = wt;
}
```

This method is not only inefficient, but even more: it may not work when the composed object is declared as a `const` object. A data field like `birthday` is a good candidate for being `const`, since a person's birthday usually doesn't change too much.

This means that when the definition of a `Person` is altered so as to contain a `string const birthday` member, the implementation of the constructor `Person::Person()` in which also the `birthday` must be initialized, a member initializer *must* be used for `birthday`. Direct assignment of the `birthday` would be illegal, since `birthday` is a `const` data member. The next example illustrates the `const` data member initialization:

```
Person::Person(char const *nm, char const *adr,
               char const *ph, char const *birth,
               unsigned wt)
:
    name(nm),
    address(adr),
    phone(ph),
    birthday(birth),    // assume: string const birthday
    weight(wt)
{ }
```

Concluding, the rule of thumb is the following: when composition of objects is used, the member initializer method is preferred to explicit initialization of composed objects. This not only results in more efficient code, but it also allows composed objects to be declared as `const` objects.

## 6.4.2 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 that another object that is used in the function `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. Passing an object in a blunt way (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:

```
Process::Process(Configfile conf)    // a copy from the caller
{
    conf_member = conf;              // copying to conf_member
}
```

- The copy-instructions can be avoided by using *pointers* to the `Configfile` objects, as in:

```
Process::Process(Configfile *conf)   // pointer to external object
{
    conf_ptr = conf;                 // conf_ptr 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* to the `Process` constructor. Next, we can define a `Config` reference data member in the class `Process`. Using the reference variable effectively uses a pointer, disguised as a variable.

However, the following construction will *not* result in the correct initialization of the `Configfile` `&conf_ref` reference data member:

```
Process::Process(Configfile &conf)
{
    conf_ref = conf;                // wrong: no assignment
}
```

The statement `conf_ref = conf` fails, because the compiler won't see this as an initialization, but considers this an assignment of one `Configfile` object (i.e., `conf`), to another (`conf_ref`). It does so, because that's the normal interpretation: an assignment to a reference variable is actually an assignment to the variable the reference variable refers to. But to what variable does `conf_ref` refer? To no variable, since we haven't initialized `conf_ref`. After all, the whole purpose of the statement `conf_ref = conf` was to initialize `conf_ref`...

So, how do we proceed when `conf_ref` must be initialized? In this situation we once again use the member initializer syntax. The following example shows the correct way to initialize `conf_ref`:

```
Process::Process(Configfile &conf)
```

```

:
    conf_ref(conf)      // initializing reference member
{}

```

Note that this syntax can be used in all cases where reference data members are used. If `int_ref` would be an `int` reference data member, a construction like

```

Process::Process(int &ir)
:
    int_ref(ir)
{}

```

would have been called for.

## 6.5 Header file organization with classes

In section 2.5.9 the requirements for header files when a C++ program also uses C functions were discussed.

When classes are used, there are more requirements for the organization of header files. In this section these requirements are covered.

First, the source files. With the exception of the occasional classless function, source files should contain the code of member functions of classes. With source files 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 are included in the class-headerfile, and each sourcefile of that class includes only the header file of its class.

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 sourcefiles: 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 will often have to read header files which aren't actually used by the function defined in the source file.

With computers running faster and faster we think the second alternative is to be preferred over the first alternative. So, as a starting point we suggest that source files of a particular class `MyClass` are organized according to the following example:

```

#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. If name collisions with existing header files might occur it pays off to have a subdirectory of one of the directories mentioned in the `INCLUDE` environment variable (e.g., `/usr/local/include/myheaders/`). If a class `MyClass` is developed there, create a subdirectory (or subdirectory link) `myheaders` of one of the standard `INCLUDE` directories to contain all header files of all classes that are developed as part of the project. The `include`-directives will then be similar to `#include <myheaders/myclass.h>`, and name collisions with other header files are avoided.

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:

```
#ifndef _String_h_
#define _String_h_

#include <project/file.h>    // to know about a File

class String
{
public:
    void getLine(File &file);
};
#endif
```

However, a similar setup is required for the class `File`:

```
#ifndef _File_h_
#define _File_h_

#include <project/string.h>    // to know about a String

class File
{
public:
    void gets(String &string);
};
#endif
```

Now we have created a problem. The compiler, trying to compile `File::gets()` proceeds as follows:

- The header file `project/string.h` is opened to be read
- `_String_h_` is defined
- 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_` has been defined, so `project/string.h` is skipped



- The definition of the `class File` is parsed.
- In the class definition contains a reference to a `String` object
- As the `class String` hasn't been parsed yet, a `String` is an undefined type, and the compiler quits with an error.

The solution for this problem is to use a *forward class reference before* the class definition, and to include the corresponding class header file *after* the class definition. So we get:

```
#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`:

```
#ifndef _File_h_
#define _File_h_

class String;              // forward reference

class File
{
public:
    void gets(String &string);
};

#include <project/string.h>  // to know about a String

#endif
```

This works well in all situations where either references or pointers to another class are involved. But it doesn't work with composition. Assume the class `File` has a *composed* data member of the class `String`. In that case, the class definition of the class `File` *must* include the header file of the class `String` before the class definition itself, because otherwise the compiler can't tell how big a `File` object will be, as it doesn't know the size of a `String` object once the definition of the `File` class is completed.

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 definition itself. In such a case the `class File` might be defined as follows:

```
#ifndef _File_h_
```

```

#define _File_h_

#include <project/string.h>    // to know about a String

class File
{
    public:
        void gets(String &string);
    private:
        String          // composition !
            line;
};
#endif

```

Note that the class `String` can't have a `File` object as a composed member: such a situation would result again in an undefined class while compiling the sources of these classes.

All other required header files are either related to classes that are used only within the source files themselves (without being part of the current class definition), or they are related to classless functions (like `memcpy()`). All headers that are not required by the compiler to parse the current class definition can be mentioned below the class definition.

This approach allows us to introduce yet another refinement:

- Header files in which a class interface is defined *declares* what can be declared before defining the class interface itself: The classes of objects appearing only as references or as pointers in the class definition are specified as forward references.

Header files of classes of objects that are either composed or inherited (see chapter 13), however, *must* be defined before the definition of the class itself starts. The information in the header file itself is protected by the `#ifndef ... #endif` construction introduced in section 2.5.9. Sources using the class 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.

- For the implementation of the member functions of the class the class header file is required and usually other header files (like `#include <string>`) are required as well. The class header file itself as well as these additional header files are included in a separate internal header file (for which the extension `.i` is proposed.<sup>2</sup>). The `.i` file should be defined in the same directory as the source files of the class, and 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`:

---

<sup>2</sup>In the past I tended to use the extension `.H` for this internal header file. However, some file systems (guess which?) have problems distinguishing lower- and uppercase filename characters. Therefore the `.i` extension is probably preferable to the `.H` extension

```

#ifndef _File_h_
#define _File_h_

#include <fstream>      // for composed 'ifstream'

class Buffer;          // forward reference

class File             // class definition
{
public:
    void gets(Buffer &buffer);
private:
    ifstream
        instream;
};
#endif

```

- Second part, e.g., ~/myproject/file/file.i, 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.

```

### 6.5.1 Using namespaces in header files

When entities from namespaces are used in header files, in general using directives should not be used in these 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 `serter` `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::serter`, then the class should be constructed as follows:

```

class special::serter;

class Flaw
{
public:
    Flaw(special::serter &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:

```

using namespace special;

```

```

class Inserter;

class Flaw
{
public:
    Flaw(Inserter &ins);
};

```

This works fine, up to the point where somebody want 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:

```

#include <flaw.h>
#include <iostream>

using std::cout;

int main()
{
    cout << "starting" << endl;    // doesn'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::Extractor`, then, because of the `using` directive in the user program, it considers `cout` a `std::ostream`. As compilers do, when confronted with an ambiguity, an error is reported.

As a rule of thumb, header files intended to be generally used should not contain `using` declarations. This rule does not hold true for header files which are included only 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.

## Chapter 7

# Classes and memory allocation

In contrast to the set of functions which handle memory allocation in C (i.e., `malloc()` etc.), the operators `new` and `delete` are specifically meant to be used with the features that C++ offers. 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 `ints` is allocated, the programmer must supply the correct expression using a multiplication by `sizeof(int)`. In contrast, `new` requires the use of a type; the `sizeof` expression is implicitly handled by the compiler.
- The only way to initialize memory which is allocated by `malloc()` is to use `calloc()`, which allocates memory and resets it to a given value. In contrast, `new` can call the constructor of an allocated object where initial actions are defined. This constructor may be supplied with arguments.
- All C-allocation functions must be inspected for NULL-returns. In contrast, the `new`-operator provides a facility called a *new\_handler* (cf. section 7.2.2) which can be used instead of the explicit checks for NULL-returns.

A comparable relationship exists between `free()` and `delete`: `delete` makes sure that when an object is deallocated, a corresponding destructor is called.

The automatic calling of constructors and destructors when objects are created and destroyed, has a number of 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 a number of handy tools.

Unfortunately, the very frequently used `str...()` functions, like `strdup()` are all `malloc()` based, and should therefore preferably not be used anymore in C++ programs. Instead, a new set of corresponding functions, based on the operator `new`, are preferred. Also, since the class `string` is available, there is less need for these functions in C++ than in C. In cases where operations on `char *` are preferred or necessary, comparable functions based on `new` could be developed. E.g., for the function `strdup()` a comparable function `char *strdupnew(char const *str)` could be developed as follows:

```
char *strdupnew(char const *str)
{
    return str ? strcpy(new char [strlen(str) + 1], str) : 0;
```

```
}
```

In this chapter the following topics will be covered:

- the assignment operator (and operator overloading in general),
- the `this` pointer,
- the copy constructor.

## 7.1 The operators ‘new’ and ‘delete’

The C++ language defines two operators which are specific for the allocation and deallocation of memory. These operators are `new` and `delete`.

The most basic example of the use of these operators is given below. An `int` pointer variable is used to point to memory which is allocated by the operator `new`. This memory is later released by the operator `delete`.

```
int
    *ip;

ip = new int;
delete ip;
```

Note that `new` and `delete` are *operators* and therefore do not require parentheses, as required for *functions* like `malloc()` and `free()`. The operator `delete` returns `void`, the operator `new` returns a pointer to the kind of memory that's asked for by its argument (e.g., a pointer to an `int` in the above example). Note that the operator `new` uses a *type* as its operand, which has the benefit that the correct amount of memory, given the type of the object to be allocated, becomes automatically available. Furthermore, this is a type safe procedure as `new` returns a pointer to the type that was given as its operand, which pointer must match the type of the variable receiving the pointer value.

The operator `new` can be used to allocate primitive types and to allocate objects. When a primitive type is allocated, the allocated memory is initialized to 0. Alternatively, an initialization expression may be provided:

```
int
    *v1 = new int,           // initialized to 0
    *v2 = new int(3),        // initialized to 3
    *v3 = new int(3 * *v2);  // initialized to 9
```

When objects are allocated, the constructor must be mentioned, and the allocated memory will be initialized according to the constructor that is used. For example, to allocate a `string` object the following statement can be used:

```
string *s = new string();
```

Here, the default constructor was used, and `s` will point to the newly allocated, but empty, `string`. If overloaded forms of the constructor are available, these can be used as well. E.g.,

```
string *s = new string("hello world");
```

which results in `s` pointing to a `string` containing the text `hello world`.

Memory allocation may fail. What happens then is unveiled in section 7.2.2.

### 7.1.1 Allocating arrays

Operator `new[]` is used to allocate arrays. The generic notation `new[]` is an abbreviation used in the Annotations. Actually, the number of elements to be allocated is specified as an expression between the square brackets, which are *prefixed* by the type of the values or class of the objects that must be allocated:

```
int *intarr = new int[20]; // allocates 20 ints
```

Note well that operator `new` is a different operator than operator `new[]`. In section 9.8 redefining operator `new[]` is covered.

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 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. With the array allocation, all elements are stored consecutively in memory. The array index notation can be used to access the individual elements: `intarr[0]` will be the very first `int` value, immediately followed by `intarr[1]`, and so on until the last element: `intarr[19]`.

To allocate arrays of objects, the `new[]`-bracket notation is used as well. For example, to allocate an array of 20 `string` objects the following construction is used:

```
string *strarr = new string[20]; // allocates 20 strings
```

Note here that, since *objects* are allocated, constructors are automatically used. So, whereas `new int[20]` results in a block of 20 *uninitialized* `int` values, `new string[20]` results in a block of 20 *initialized* `string` objects. With arrays of objects the *default constructor* is used for the initialization. Unfortunately it is not possible to use a constructor having arguments when arrays of objects are allocated. However, it is possible to *overload* operator `new[]` and provide it with arguments which may be used for a non-default initialization of arrays of objects. Overloading operator `new[]` is discussed in section 9.8.

Similar to C, and without resorting to the operator `new[]`, arrays of variable size can also be constructed as *ilocal arrays* within functions. Such arrays are not dynamic arrays, but *local arrays*, and their lifetime is restricted to the lifetime of the block in which they were defined.

Once allocated, all arrays are fixed size arrays. There is no simple way to enlarge or shrink arrays: there is no *renew* operator. In section 7.1.3 an example is given showing how to enlarge an array.

### 7.1.2 Deleting arrays

A dynamically allocated array may be deleted using operator `delete[]`. Operator `delete[]` expects a pointer to a block of memory, previously allocated using operator `new[]`.

When an object is deleted, its *destructor* (see section 7.2) is called automatically, comparably to the calling of the object's constructor when the object was created. It is the task of the destructor, as discussed in depth later in this chapter, to do all kinds of cleanup operations that are required for the proper destruction of the object.

The operator `delete[]` (empty square brackets) expects as its argument a pointer to an array of objects. This operator will now first call the destructors of the individual objects, and will then delete the allocated block of memory. So, the proper way to delete an array of `Object`s is:

```
Object
    *op = new Object[10];
delete[] op;
```

Realize that `delete[]` only has an effect if the block of memory to be deallocated contains *objects*. With any other type of element normally no special action is performed: following `int *it = new int[10]` the statement `delete[] it` the memory occupied by all ten `int` values is returned to the common pool. Nothing special happens.

Note especially that an array of pointers to objects is not handled as an array of objects by `delete[]`: the array of pointers to objects doesn't contain objects, so the objects are not properly destroyed by `delete[]`, whereas an array of objects contains objects, which are properly destroyed by `delete[]`. In section 7.2 several examples of the use of `delete` versus `delete[]` will be given.

The operator `delete` is a different operator than operator `delete[]`. In section 9.8 redefining `delete[]` is discussed. For now, the rule of thumb is: if `new[]` was used, also use `delete[]`.

### 7.1.3 Enlarging arrays

Once allocated, all arrays are fixed size arrays. There is no simple way to enlarge or shrink arrays: there is no `renew` operator. In this section an example is given showing how to enlarge an array. Enlarging arrays is only possible with dynamic arrays. Local and global arrays cannot be enlarged. When an array must be enlarged, the following procedure can be used:

- Allocate a new block of memory, of larger size
- Copy the old array contents to the new array
- Delete the old array (see section 7.1.2)
- Have the old array pointer point to the newly allocated array

The following example focuses on the enlargement of an array of `string` objects:

```
#include <string>

string *enlarge(string *old, unsigned oldsize, unsigned newsize)
{
    string
        *tmp = new string[newsize];    // allocate larger array

    for (unsigned idx = 0; idx < oldsize; ++idx)
        tmp[idx] = old[idx];           // copy old to tmp
```



```

        delete[] old;                // delete old, using [] due to objects

        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.
}

```

## 7.2 The destructor

Comparable to the constructor, classes may define a *destructor*. This function is the opposite of the constructor in the sense that it is invoked when an object ceases to exist. For objects which are local non-static variables, the destructor is called when the block in which the object is defined is left: the destructors of objects that are defined in nested blocks of functions are therefore usually called before the function itself terminates. The destructors of objects that are defined somewhere in the outer block of a function are called just before the function returns (terminates). For static or global variables the destructor is called before the program terminates.

However, when a program is interrupted using an `exit()` call, the destructors are called *only* for global objects which exist at that time. Destructors of objects defined *locally* within functions are not called when a program is forcefully terminated using `exit()`.

The definition of a destructor must obey the following rules:

- The destructor has the same name as the class but its name is prefixed by a tilde.
- The destructor has no arguments and no a return value.

The destructor for the class `Person` could thus be declared as follows:

```

class Person
{
    public:
        Person();                // constructor
        ~Person();               // destructor
};

```

The position of the constructor(s) and destructor in the class definition is dictated by convention: First the constructors are declared, then the destructor, and only then any other members follow.

The main task of a destructor is to make sure that memory allocated by the object (e.g., by its constructor) is properly deleted when the object goes out of scope. Consider the following definition of the class `Person`:

```

class Person
{

```

```

public:
    Person()
    {}
    Person(char const *n, char const *a,
           char const *p);
    ~Person();

    char const *getName() const;
    char const *getAddress() const;
    char const *getPhone() const;

private:
    // data fields
    char *name;
    char *address;
    char *phone;
};

```

The task of the constructor is to initialize the data fields of the object. E.g, the constructor is defined as follows:

```

#include "person.h"
#include <string.h>

Person::Person(char const *n, char const *a, char const *p)
{
    name    = strdupnew(n);
    address = strdupnew(a);
    phone   = strdupnew(p);
}

```

In this class the destructor is necessary to prevent that memory, once allocated for the fields `name`, `address` and `phone`, becomes unreachable when an object ceases to exist, thus producing a memory leak. The destructor of an object is called automatically

- When an object goes out of scope;
- When a dynamically allocated object is deleted;
- When an dynamically allocated array of objects is deleted using the `delete[]` operator (see section 7.1.2).

Since it is the task of the destructor to delete all memory that was dynamically allocated and used by the object, the task of the `Person`'s destructor would be to delete the memory pointed to by its three data members. The implementation of the destructor would therefore be:

```

#include "person.h"
#include <string.h>

Person::~~Person()
{
    delete name;
}

```

```

        delete address;
        delete phone;
    }

```

In the following example a `Person` object is created, and its data fields are printed. After this the `showPerson()` function stops, which leads to the deletion of memory. Note that in this example a second object of the class `Person` is created and destroyed dynamically by respectively, the operators `new` and `delete`.

```

#include "person.h"
#include <iostream>

void showPerson()
{
    Person
        karel("Karel", "Marskramerstraat", "038 420 1971"),
        *frank = new Person("Frank", "Oostumerweg", "050 403 2223");

    cout << karel.getName()    << ", " <<
         karel.getAddress()   << ", " <<
         karel.getPhone()     << endl <<
         frank->getName()     << ", " <<
         frank->getAddress()  << ", " <<
         frank->getPhone()    << endl;

    delete frank;
}

```

The memory occupied by the object `karel` is deleted automatically when `showPerson()` terminates: the C++ compiler makes sure that the destructor is called. Note, however, that the object pointed to by `frank` is handled differently. The variable `frank` is a pointer, and a pointer variable is itself no `Person`. Therefore, before `main()` terminates, the memory occupied by the object pointed to by `frank` should be *explicitly* deleted; hence the statement `delete frank`. The operator `delete` will make sure that the destructor is called, thereby deleting the three `char *` strings of the object.

### 7.2.1 New and delete and object pointers

The operators `new` and `delete` are used when an object of a given class is allocated. As we have seen, the advantage of the operators `new` and `delete` over functions like `malloc()` and `free()` lies in the fact that `new` and `delete` call the corresponding constructors or destructor. This is illustrated in the next example:

```

Person
    *pp = new Person();    // ptr to Person object

delete pp;                // now destroyed

```

The allocation of a new `Person` object pointed to by `pp` is a two-step process. First, the memory for the object itself is allocated. Second, the constructor is called which initializes the object. In the above example the constructor is the argument-free version; it is however also possible to use a constructor having arguments:

```
frank = new Person("Frank", "Oostumerweg", "050 403 2223");
delete frank;
```

Note that, analogously to the *construction* of an object, the *destruction* is also a two-step process: first, the destructor of the class is called to delete the memory allocated and used by the object; then the memory which is used by the object itself is freed.

Dynamically allocated arrays of objects can also be manipulated by `new` and `delete`. In this case the size of the array is given between the `[]` when the array is created:

```
Person
    *personarray = new Person [10];
```

The compiler will generate code to call the default constructor for each object which is created. As we have seen in section 7.1.2, the `delete[]` operator must here be used to destroy such an array in the proper way:

```
delete[] personarray;
```

The presence of the `[]` ensures that the destructor is called for each object in the array.

What happens if `delete` rather than `delete[]` is used? Consider the following situation, in which the destructor `~Person()` is modified so that it will tell us that it's called. In a `main()` function an array of two `Person` 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 "person.h"

Person::~~Person()
{
    cout << "Person destructor called" << endl;
}

int main()
{
    Person
        *a = new Person[2];

    cout << "Destruction with []'s" << endl;

    delete [] a;

    a = new Person[2];

    cout << "Destruction without []'s" << endl;

    delete a;

    return 0;
}
/*
Generated output:
```

```

Destruction with []'s
Person destructor called
Person destructor called
Destruction without []'s
Person destructor called
*/

```

Looking at the generated output, we see that the destructor of the individual `Person` objects are called if the `delete[]` syntax is followed, and not if the `[]` is omitted.

If no destructor is defined, it is not called. This may seem to be a trivial statement, but it has severe implications: objects which allocate memory will result in a memory leak when no destructor is defined. Consider the following program:

```

#include <iostream>
#include "person.h"

Person::~~Person()
{
    cout << "Person destructor called" << endl;
}

int main()
{
    Person
        **a;

    a = new Person* [2];

    a[0] = new Person [2];
    a[1] = new Person [2];

    delete [] a;

    return 0;
}

```

This program produces no output at all. Why is this? The variable `a` is defined as a *pointer to a pointer*. For this situation, however, there is no defined destructor. Consequently, the `[]` is ignored.

Now, because of the `[]` being ignored, only the array `a` itself is deleted, because here '`delete[] a`' deletes the memory pointed to by `a`. That's all there is to it.

Of course, we don't want this, but require the `Person` objects pointed to by the elements of `a` to be deleted too. In this case we have two options:

- Explicitly walk all the elements of the `a` array, deleting them in turn. This will call the destructor for a pointer to `Person` objects, which will destroy all elements if the `[]` operator is used, as in:

```

#include <iostream>
#include "person.h"

```

```

Person::~Person()
{
    cout << "Person destructor called" << endl;
}

int main()
{
    Person
        **a;

    a = new Person* [2];

    a[0] = new Person [2];
    a[1] = new Person [2];

    for (int index = 0; index < 2; index++)
        delete [] a[index];

    delete[] a;
}
/*
    Producing output:
Person destructor called
Person destructor called
Person destructor called
Person destructor called
*/

```

- Define a wrapper class containing a pointer to Person objects, and allocate a pointer to this class, rather than a pointer to a pointer to Person objects. The topic of containing classes in classes, *composition*, was discussed in section 6.4. Here is an example showing the deletion of pointers to memory using such a wrapper class:

```

#include <iostream>

class Informer
{
public:
    ~Informer()
    {
        cout << "destructor called\n";
    }
};

class Wrapper
{
public:
    Wrapper()
    :
        i(new Informer())
    {}
    ~Wrapper()
    {
        delete i;
    }
}

```

```

        Informer
            *i;
};

int main()
{
    delete [] new Informer *[4];    // memory leak: no destructor called

    cout << "=====\n";

    delete [] new Wrapper[4];      // ok: 4 x destructor called
}
/*
    Generated output:
    =====
    destructor called
    destructor called
    destructor called
    destructor called
    */

```

### 7.2.2 The function `set_new_handler()`

The C++ run-time system makes sure that when memory allocation fails, an error function is activated. By default this function returns the value 0 to the caller of `new`, so that the pointer which is assigned by `new` is set to zero. The error function can be redefined, but it must comply with a few prerequisites:

- it has no arguments, and
- it returns no value

Please make sure you use this function: it saves you a lot of checks (and problems with a failing allocation that you just happened to forget to protect with a check...).

The 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()`, defined in the header file `new.h`.

The implementation of an error function is illustrated below. This implementation applies to the Gnu C/C++ requirements<sup>1</sup>:

```

#include <new.h>
#include <iostream>

void outOfMemory()
{
    cout << "Memory exhausted. Program terminates." << endl;
    exit(1);
}

```

---

<sup>1</sup> The actual try-out of the program is not encouraged, as it will slow down the computer enormously due to the resulting occupation of Unix' *swap area*

```

int main()
{
    long
        allocated = 0;

    set_new_handler(outOfMemory);        // install error function

    while (1)                            // eat up all memory
    {
        new int [10000];
        allocated += 10000 * sizeof(int);
        cout << "Allocated " << allocated << " bytes\n";
    }
    return 0;
}

```

The advantage of an allocation error function lies in the fact that once installed, `new` can be used without wondering whether the allocation succeeded or not: upon failure the error function is automatically invoked and the program exits. It is good practice to install a `new` handler in each C++ program, even when the actual code of the program does not allocate memory. Memory allocation can also fail in not directly visible code, e.g., when streams are used or when strings are duplicated by low-level functions.

Note that it may *not* be assumed that the standard C functions which allocate memory, such as `strdup()`, `malloc()`, `realloc()` etc. will trigger the `new` handler when memory allocation fails. This means that once a `new` handler is installed, such functions should not automatically be used in an unprotected way in a C++ program. As an example of the use of `new` to duplicate a string, a rewrite of the function `strdup()` using the operator `new` is given in section 7. It is strongly advised to revert to this approach, rather than to keep using functions like `strdup()`, when the allocation of memory is required.

## 7.3 The assignment operator

Variables which are `structs` or `classes` can be directly assigned in C++ in the same way that `structs` can be assigned in C. The default action of such an assignment is a straight bitwise copy from one compound variable to another. Now consider the consequences of this default action in a function such as the following:

```

void printperson(Person const &p)
{
    Person
        tmp;

    tmp = p;
    cout << "Name:      " << tmp.getName()      << endl <<
         "Address:   " << tmp.getAddress()    << endl <<
         "Phone:     " << tmp.getPhone()      << endl;
}

```

We shall follow the execution of this function step by step.

- The function `printperson()` expects a reference to a `Person` as its parameter `p`. So far, nothing



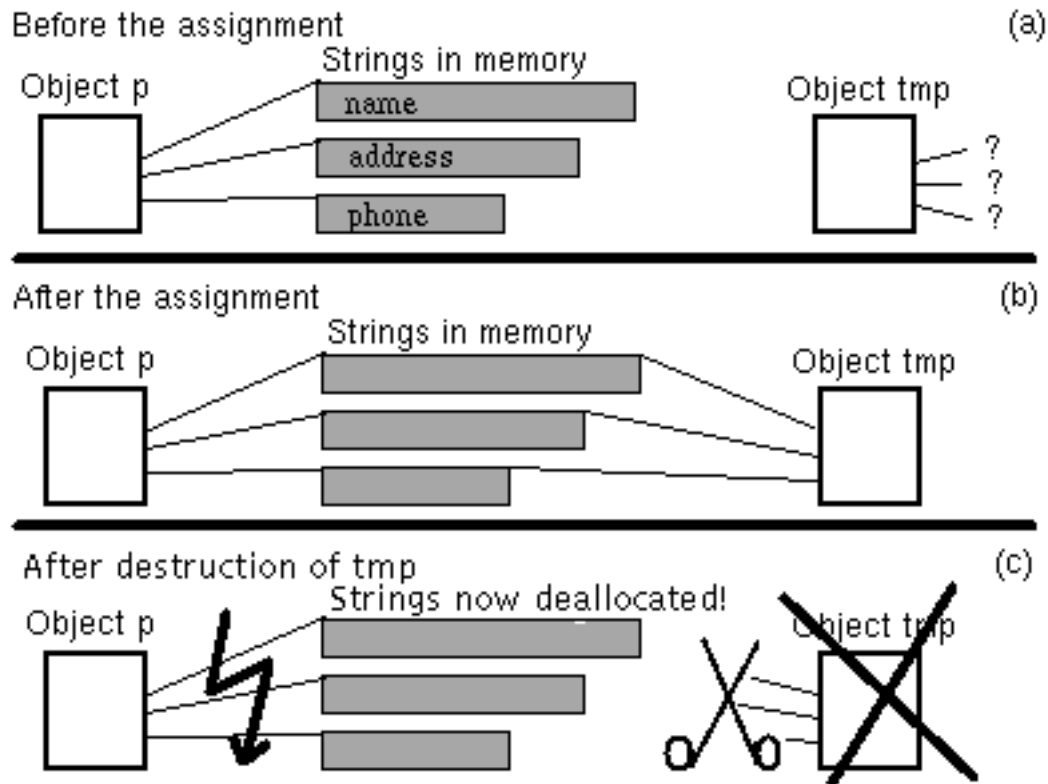


Figure 7.1: Private data and public interface functions of the class `Person`, using bitwise assignment

extraordinary is happening.

- The function defines a local object `tmp`. This means that the default constructor of `Person` is called, which -if defined properly- resets the pointer fields `name`, `address` and `phone` of the `tmp` object to zero.
- Next, the object referenced by `p` is copied to `tmp`. By default this means that `sizeof(Person)` bytes from `p` are copied to `tmp`.

Now a potentially dangerous situation has arisen. Note that the actual values in `p` are *pointers*, pointing to allocated memory. Following the assignment this memory is addressed by two objects: `p` and `tmp`.

- The potentially dangerous situation develops into an acutely dangerous situation when the function `printperson()` terminates: the object `tmp` is destroyed. The destructor of the class `Person` releases the memory pointed to by the fields `name`, `address` and `phone`: unfortunately, this memory is also in use by `p`.... The incorrect assignment is illustrated in figure 7.1.

Having executed `printperson()`, the object which was referenced by `p` now contain pointers to deleted memory.

This situation is undoubtedly not a desired effect of a function like the above. The deleted memory will likely become occupied during subsequent allocations: the pointer members of `p` 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.*

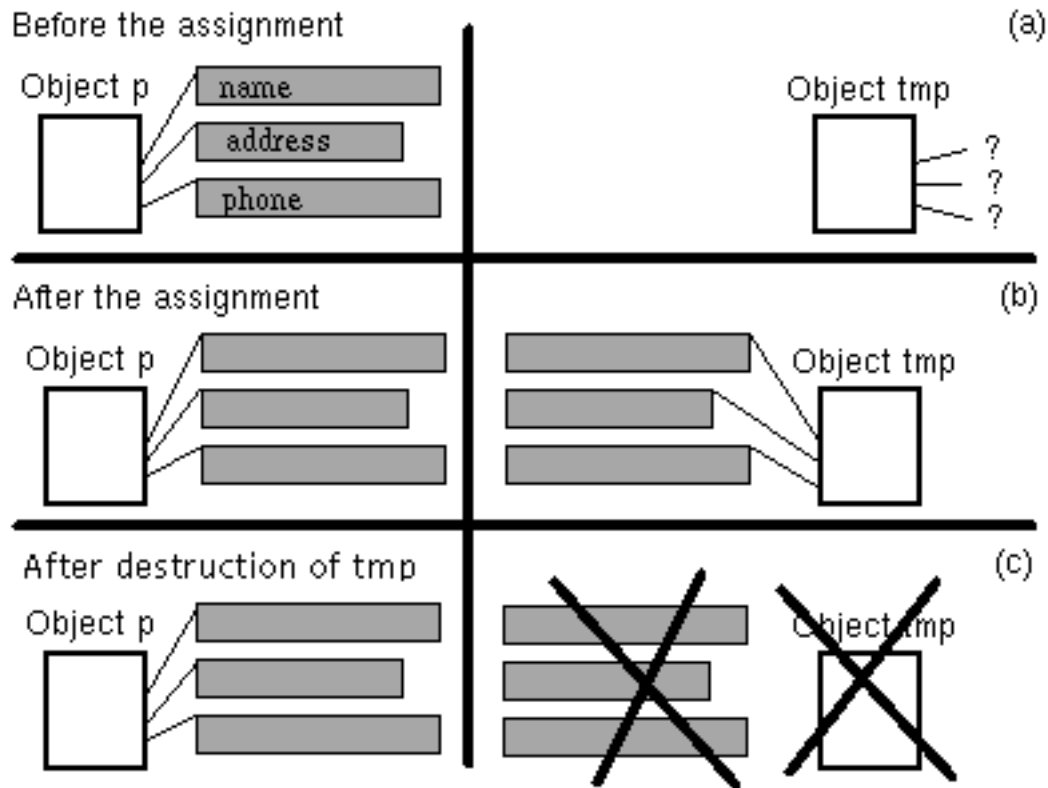


Figure 7.2: Private data and public interface functions of the class `Person`, using the 'correct' assignment.

Fortunately, it is possible to prevent these troubles, as discussed in the next section.

### 7.3.1 Overloading the assignment operator

Obviously, the right way to assign one `Person` object to another, is **not** to copy the contents of the object byte-wise. A better way is to make an equivalent object; one with its own allocated memory, but which contains the same strings.

The 'right' way to duplicate a `Person` object is illustrated in figure 7.2.

There are several ways to duplicate a `Person` object. One way would be to define a special function to handle assignments of objects of the class `Person`. The purpose of this function would be to create a copy of an object, but one with its own name, address and phone strings. Such a member function might be:

```
void Person::assign(Person const &other)
{
    // delete our own previously used memory
    delete name;
    delete address;
    delete phone;

    // now copy the other Person's data
```

```

    name = strdupnew(other.name);
    address = strdupnew(other.address);
    phone = strdupnew(other.phone);
}

```

Using this tool we could rewrite the offending function `printperson()`:

```

void printperson(Person const &p)
{
    Person
        tmp;

    // make tmp a copy of p, but with its own allocated memory
    tmp.assign(p);

    cout << "Name:      " << tmp.getname()          << endl <<
         "Address:   " << tmp.getaddress()         << endl <<
         "Phone:     " << tmp.getphone()           << endl;

    // now it doesn't matter that tmp gets destroyed..
}

```

In itself this solution is valid, although it is a purely symptomatic solution. This solution requires the programmer to use a specific member function instead of the operator `=`. The basic problem, however, remains if this rule is not strictly adhered to. Experience learns that *errare humanum est*: a solution which doesn't enforce special actions is therefore preferable.

The problem of the assignment operator is solved using *operator overloading*: the syntactic possibility C++ offers to redefine the actions of an operator in a given context. Operator overloading was mentioned earlier, when the operators `<<` and `>>` were redefined for the usage with streams as `cin`, `cout` and `cerr` (see section 3.1.2).

Overloading the assignment operator is probably the most common form of operator overloading. However, a word of warning is appropriate: the fact that C++ allows operator overloading does not mean that this feature should be used at all times. A few rules are:

- Operator overloading should be used in situations where an operator has a defined action, but when this action is not desired as it has negative side effects. A typical example is the above assignment operator in the context of the class `Person`.
- Operator overloading can be used in situations where the use of the operator is common and when no ambiguity in the meaning of the operator is introduced by redefining it. An example may be the redefinition of the operator `+` for a class which represents a complex number. The meaning of a `+` between two complex numbers is quite clear and unambiguous.
- In all other cases it is preferable to define a member function, instead of redefining an operator.

Using these rules, operator overloading is minimized which helps keep source files readable. An operator simply does what it is designed to do. Therefore, in our vision, the insertion (`<<`) and extraction (`>>`) operators in the context of streams are unfortunate: the stream operations do not have anything in common with the bitwise shift operations.

### The function 'operator=()'

To achieve operator overloading in the context of a class, the class is simply expanded with a public function stating the particular operator. A corresponding function, the implementation of the overloaded operator, is thereupon defined.

For example, to overload the addition operator +, a function operator+() must be defined. The function name consists of two parts: the keyword operator, followed by the operator itself.

In our case we define a new function operator=() to redefine the actions of the assignment operator. A possible extension to the class Person could therefore be:

```
class Person
{
    public:                                // extension of the class Person
                                           // earlier members are assumed.
    void operator=(Person const &other);
};
```

and the implementation could be

```
void Person::operator=(Person const &other)
{
    delete name;                          // delete old data
    delete address;
    delete phone;

    name = strdupnew(other.name);          // duplicate other's data
    address = strdupnew(other.address);
    phone = strdupnew(other.phone);
}
```

The function operator=() presented here is the first version of the overloaded assignment operator. We shall present improved versions shortly.

The actions of this member function are similar to those of the previously proposed function assign(), but now its *name* makes sure that this function is also activated when the assignment operator = is used. There are actually two ways to call overloaded operators:

```
Person
    pers("Frank", "Oostumerweg", "403 2223"),
    copy;

copy = pers;                // first possibility
copy.operator=(pers);       // second possibility
```

It is obvious that the second possibility, in which operator=() is explicitly stated, is not used often. However, the code fragment *does* illustrate the two ways of calling the same function.

## 7.4 The this pointer

As we have seen, a member function of a given class is always called in the context of some object of the class. There is always an implicit 'substrate' for the function to act on. C++ defines a keyword, `this`, to address this substrate<sup>2</sup>

The `this` keyword is a pointer variable, which always contains the address of the object in question. The `this` pointer is implicitly declared in each member function (whether public, protected or private). Therefore, it is as if each member function of the class `Person` contains the following declaration:

```
extern Person *this;
```

A member function like `getName()`, which returns the `name` field of a `Person`, could therefore be implemented in two ways: with or without the `this` pointer:

```
char const *Person::getName() // implicit usage of 'this'
{
    return name;
}

char const *Person::getName() // explicit usage of 'this'
{
    return this->name;
}
```

Explicit usage of the `this` pointer is not used very frequently. However, several situations exist where the `this` pointer is really needed.

### 7.4.1 Preventing self-destruction with this

As we have seen, the operator `=` can be redefined for the class `Person` in such a way that two objects of the class can be assigned, resulting in two copies of the same object.

As long as the two variables are different ones, the previously presented version of the function `operator=()` will behave properly: the memory of the assigned object is released, after which it is allocated again to hold new strings. However, when an object is assigned to itself (which is called *auto-assignment*), a problem occurs: the allocated strings of the receiving object are first deleted, resulting in the deletion of the memory of the right-hand side variable, which we call *self-destruction*. An example of this situation is illustrated here:

```
void fubar(Person const &p)
{
    p = p;           // auto-assignment!
}
```

In this example it is perfectly clear that something unnecessary, possibly even wrong, is happening. But auto-assignment can also occur in more hidden forms:

`Person`

---

<sup>2</sup>Note that '`this`' is not available in the not yet discussed static member functions.

```

    one,
    two,
    *pp = &one;

*pp = two;
one = *pp;

```

The problem of auto-assignment can be solved using the `this` pointer. In the overloaded assignment operator function we simply test whether the address of the right-hand side object is the same as the address of the current object: if so, no action needs to be taken. The definition of the function `operator=()` thus becomes:

```

void Person::operator=(Person const &other)
{
    // only take action if address of the current object
    // (this) is NOT equal to the address of the other object

    if (this != &other)
    {
        delete name;
        delete address;
        delete phone;

        name = strdupnew(other.name);
        address = strdupnew(other.address);
        phone = strdupnew(other.phone);
    }
}

```

This is the second version of the overloaded assignment function. One, yet better version remains to be discussed.

As a subtlety, note the usage of the *address operator* `'&'` in the statement

```

if (this != &other)

```

The variable `this` is a pointer to the 'current' object, while `other` is a reference; which is an 'alias' to an actual `Person` object. The address of the other object is therefore `&other`, while the address of the current object is `this`.

## 7.4.2 Associativity of operators and `this`

According to C++'s syntax, the associativity of the assignment operator is to the right-hand side. I.e., in statements like:

```

a = b = c;

```

the expression `b = c` is evaluated first, and the result is assigned to `a`.

The implementation of the overloaded assignment operator so far does not permit such constructions, as an assignment using the member function returns nothing (`void`). We can therefore conclude that the previous implementation does solve an allocation problem, but still prevents concatenated assignments.

The problem can be illustrated as follows. When we rewrite the expression `a = b = c` to the form which explicitly mentions the overloaded assignment member functions, we get:

```
a.operator=(b.operator=(c));
```

This variant is syntactically wrong, since the sub-expression `b.operator=(c)` yields `void`. However, the class `Person` contains no member functions with the prototype `operator=(void)`.

This problem too can be remedied using the `this` pointer. The overloaded assignment function expects as its argument a reference to a `Person` object. It can also *return* a reference to such an object. This reference can then be used as an argument for a concatenated assignment.

It is customary to let the overloaded assignment return a reference to the current object (i.e., `*this`). The (final) version of the overloaded assignment operator for the class `Person` thus becomes:

```
Person &Person::operator=(Person const &other)
{
    if (this != &other)
    {
        delete address;
        delete name;
        delete phone;

        address = strdupnew(other.address);
        name = strdupnew(other.name);
        phone = strdupnew(other.phone);
    }
    // return current object. The compiler will make sure
    // that a reference is returned
    return *this;
}
```

## 7.5 The copy constructor: Initialization vs. Assignment

In the following sections we shall take a closer look at another usage of the operator `=`. Consider, once again, the class `Person`. The class has the following characteristics:

- The class contains several pointers, possibly pointing to allocated memory. As discussed, such a class needs a constructor and a destructor.  
A typical action of the constructor would be to set the pointer members to 0. A typical action of the destructor would be to delete the allocated memory.
- For the same reason the class requires an overloaded assignment operator.
- The class has, besides a default constructor, a constructor which expects the name, address and phone number of the `Person` object.
- For now, the only remaining interface functions return the name, address or phone number of the `Person` object.

Now consider the following code fragment. The statement references are discussed following the example:

```

Person
    karel("Karel", "Marskramerstraat", "038 420 1971"), // see (1)
    karel2, // see (2)
    karel3 = karel; // see (3)

int main()
{
    karel2 = karel3 // see (4)
    return 0;
}

```

- Statement 1: this statement shows an initialization. The object `karel` is initialized with appropriate texts. This construction of the object `karel` therefore uses the constructor which expects three `char const *` arguments.

Assume a `Person` constructor is available having only one `char const *` parameter, e.g., `Person::Person(char const *)`. It should be noted that the initialization '`Person frank("Frank")`' is identical to

```

Person
    frank = "Frank";

```

Even though this piece of code uses the operator `=`, it is no assignment: rather, it is an *initialization*, and hence, it's done at *construction time* by a constructor of the class `Person`.

- Statement 2: here a second `Person` object is created. Again a constructor is called. As no special arguments are present, the *default constructor* is used.
- Statement 3: again a new object `karel3` is created. A constructor is therefore called once more. The new object is also initialized. This time with a copy of the data of object `karel`. This form of initializations has not yet been discussed. As we can rewrite this statement in the form

```

Person
    karel3(karel);

```

it is suggested that a constructor is called, having a reference to a `Person` object as its argument. Such constructors are quite common in C++ and are called *copy constructors*. More properties of these constructors are discussed below.

- Statement 4: here one object is assigned to another. No object is *created* in this statement. Hence, this is just an assignment, using the overloaded assignment operator.

The simple rule emanating from these examples is that *whenever an object is created, a constructor is needed*. All constructors have the following characteristics:

- Constructors have no return values.
- Constructors are defined in functions having the same names as the class to which they belong.
- The argument list of constructors can be deduced from the code. The argument is either present between parentheses or (if there is only one argument) following a `=`.

Therefore, we conclude that, given the above statement (3), the class `Person` must be augmented with a *copy constructor*:

```

class Person

```



```

{
    public:
        Person(Person const &other);
};

```

The implementation of the `Person` copy constructor is:

```

Person::Person(Person const &other)
{
    name    = strdupnew(other.name);
    address = strdupnew(other.address);
    phone   = strdupnew(other.phone);
}

```

The actions of copy constructors are comparable to those of the overloaded assignment operators: an object is *duplicated*, so that it contains its own allocated data. The copy constructor, however, is simpler in the following respects:

- A copy constructor doesn't need to delete previously allocated memory: since the object in question has just been created, it cannot already have its own allocated data.
- A copy constructor never needs to check whether auto-duplication occurs. No variable can be initialized with itself.

Besides the above mentioned quite obvious usage of the copy constructor, the copy constructor has other important tasks. All of these tasks are related to the fact that the copy constructor is always called when an object is created and initialized with another object of its class. The copy constructor is called even when this new object is a hidden or is a temporary variable.

- When a function takes an object as argument, instead of, e.g., a pointer or a reference, the copy constructor is called to pass a copy of an object as the argument. This argument, which usually is passed via the stack, is therefore a new object. It is created and initialized with the data of the passed argument. This is illustrated in the following code fragment:

```

void nameOf(Person p)          // no pointer, no reference
{                               // but the Person itself
    cout << p.getName() << endl;
}

int main()
{
    Person
        frank("Frank");

    nameOf(frank);
    return 0;
}

```

In this code fragment `frank` itself is not passed as an argument, but instead a temporary (stack) variable is created using the copy constructor. This temporary variable is known inside `nameOf()` as `p`. Note that if `nameOf()` would have had a reference parameter, extra stack usage and a call to the copy constructor would have been avoided.

- The copy constructor is also implicitly called when a function returns an object:

```

Person getPerson()
{
    string
        name,
        address,
        phone;

    cin >> name >> address >> phone;

    Person
        p(name.c_str(), address.c_str(), phone.c_str());

    return p;           // returns a copy of 'p'.
}

```

Here a hidden object of the class `Person` is initialized, using the copy constructor, as the value returned by the function. The local variable `p` itself ceases to exist when `getPerson()` terminates.

To demonstrate that copy constructors are not called in all situations, consider the following. We could rewrite the above function `getline()` to the following form:

```

Person getPerson()
{
    string
        name,
        address,
        phone;

    cin >> name >> address >> phone;

    return Person(name.c_str(), address.c_str(), phone.c_str());
}

```

This code fragment is perfectly valid, and illustrates the use of an anonymous object. Anonymous objects are *const objects*: their data members may not change. The use of an anonymous object in the above example illustrates the fact that object return values should be considered constant objects, even though the keyword `const` is not explicitly mentioned in the return type of the function (as in `Person const getPerson()`).

As an other example, once again assuming the availability of a `Person(char const *name)` constructor, consider:

```

Person getNamedPerson()
{
    string
        name;

    cin >> name;

    return name.c_str();
}

```

Here, even though the return value `name.c_str()` doesn't match the return type `Person`, there is a *constructor* available to construct a `Person` from a `char const *`. Since such a constructor is available, the (anonymous) return value can be constructed by *promoting* a `char const *` type to a `Person` type using an appropriate constructor.

Contrary to the situation we encountered with the default constructor, the default copy constructor remains available once a constructor (*any constructor*) is defined explicitly. The copy constructor can be redefined, but it will not disappear once another constructor is defined.

### 7.5.1 Similarities between the copy constructor and operator=()

The similarities between on the one hand the copy constructor and on the other hand the overloaded assignment operator are reinvestigated in this section. We present here two primitive functions which often occur in our code, and which we think are quite useful. Note the following features of copy constructors, overloaded assignment operators, and destructors:

- The *copying of (private) data* occurs (1) in the copy constructor and (2) in the overloaded assignment function.
- The *deletion of allocated memory* occurs (1) in the overloaded assignment function and (2) in the destructor.

The above two actions (duplication and deletion) can be coded in two private functions, say `copy()` and `destroy()`, which are used in the overloaded assignment operator, the copy constructor, and the destructor. When we apply this method to the class `Person`, we can implement this approach as follows:

- First, the class definition is expanded with two private functions `copy()` and `destroy()`. The purpose of these functions is to copy the data of another object or to delete the memory of the current object *unconditionally*. Hence these functions implement 'primitive' functionality:

```
// class definition, only relevant functions are shown here
class Person
{
    public:
        Person(Person const &other);
        ~Person();
        Person &operator=(Person const &other);
    private:
        void copy(Person const &other);    // new members
        void destroy(void);

        char
            *name,
            *address,
            *phone;
};
```

- Next, the functions `copy()` and `destroy()` are constructed:

```
void Person::copy(Person const &other)
{
    name = strdupnew(other.name);    // unconditional copying
```

```

        address = strdupnew(other.address);
        phone = strdupnew(other.phone);
    }

    void Person::destroy()
    {
        delete name;                // unconditional deletion
        delete address;
        delete phone;
    }

```

- Finally the public functions in which other object's memory is copied or in which memory is deleted are rewritten:

```

    Person::Person (Person const &other)    // copy constructor
    {
        copy(other);
    }

    Person::~~Person()                    // destructor
    {
        destroy();
    }

                                // overloaded assignment
    Person const &Person::operator=(Person const &other)
    {
        if (this != &other)
        {
            destroy();
            copy(other);
        }
        return *this;
    }

```

What we like about this approach is that the destructor, copy constructor and overloaded assignment functions are now completely standard: they are *independent* of a particular class, and *their implementations can therefore be used in every class*. Any class dependencies are reduced to the implementations of the private member functions `copy()` and `destroy()`.

Note, that the `copy()` member function is responsible for the copying of the other object's data fields to the current object. We've shown the situation in which a class *only* has pointer data members. In most situations classes have non-pointer data members as well. These members must be copied in the copy constructor as well. This can simply be realized in the copy constructor *except* for the reference data members, which *must* be initialized using the member initializer method, introduced in section 6.4.2. However, in this case the overloaded assignment operator can't be fully implemented: once initialized, a reference member cannot be given an other value, so an existing object having reference data members is inseparately attached to its referenced object(s).

### 7.5.2 Preventing the use of certain member functions

As we've seen in the previous section, situations may be encountered in which a member function can't do its job in a completely satisfactory way. In particular: an overloaded assignment operator cannot to its job completely if its class contains reference data members. In this and comparable

situations the programmer might want to *prevent* the (accidental) use of certain member functions. This can be realized in the following ways:

- Move all member functions that should not be callable to the `private` section of the class interface. This will effectively prevent the user from the class to use these members. By moving the assignment operator to the private section, objects of the class cannot be assigned to each other anymore. Here the *compiler* will detect the use of a private member outside of its class and will flag a compilation error.
- The above solution still allows the *constructor* of the class to use the unwanted member functions within the class members itself. If that is deemed undesirable as well, such functions should still be moved to the private section of the class interface, but they should not be implemented. The *compiler* won't be able to prevent the (accidental) use of these forbidden members, but the *linker* won't be able to solve the associated external reference.
- It is *not* always a good idea to *omit member functions* that should not be called from the class interface. In particular, the overloaded assignment operator has a *default* implementation that will be used if no overloaded version is mentioned in the class interface. So, with the overloaded assignment operator in particular the previously mentioned approach should be followed. Moving certain constructors to the private section of the class interface is also a good technique to prevent their use by 'the general public'.

## 7.6 Conclusion

Two important extensions to classes have been discussed in this chapter: the overloaded assignment operator and the copy constructor. As we have seen, classes with pointer data which address allocated memory are potential sources of memory leaks. The two introduced extensions represent the standard way to prevent these memory leaks.

The conclusion is therefore: as soon as a class is defined in which pointer data members are used, a *destructor*, an *overloaded assignment operator* and a *copy constructor* should be implemented.

## Chapter 8

# Exceptions

In C there are several ways to have a program react to situations which break the normal unhampered flow of the 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....
- The function may use a combination of the functions `setjmp()` and `longjmp()` 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 the above ways to handle flow-breaking situations are still available. However, the last way, using `setjmp()` and `longjmp()` isn't often seen in C++ (or even in C) programs, due to the fact that the program flow is completely disrupted.

In C++ the alternative to using `setjmp()` and `longjmp()` are *exceptions*. Exceptions implement a mechanism by which a controlled non-local exit is realized within the context of a C++ program, without the disadvantages of `longjmp()` and `setjmp()`.

Exceptions are the proper way to bail out of a situation which cannot be handled easily by a function itself, but which are not disastrous enough for the program to terminate completely. Also, exceptions provide a flexible layer of flow control between the short-range `return` and the crude `exit()`.

In this chapter the use of exceptions and their syntax will be discussed. First an example of the different impacts exceptions and `setjmp()` and `longjmp()` have on the program will be given. Then the discussion will dig into the formalities of using exceptions.

## 8.1 Using exceptions: syntax elements

With exceptions the following syntactical elements are encountered:

- **try:** The `try`-block surrounds statements in which exceptions may be generated (the parlance is for exceptions to be thrown). Example:

```
try
{
    // statements in which exceptions may be thrown
}
```

- **throw:** followed by an expression of a certain type, throws the value of the expression as an exception. The `throw` statement must be executed somewhere within the `try`-block: either directly or from within a function called directly or indirectly from the `try`-block. Example:

```
throw "This generates a char * exception";
```

- **catch:** Immediately following the `try`-block, the `catch`-block receives the thrown exceptions. Example of a `catch`-block receiving `char *` exceptions:

```
catch (char *message)
{
    // statements in which the thrown char * exceptions are handled
}
```

## 8.2 An example using exceptions

In the next two sections the same basic program will be used. The program uses two classes, `Outer` and `Inner`. An `Outer` object is created in the `main()` function, and the function `Outer::fun()` is called. Then, in the `Outer::fun()` member an `Inner` object is constructed. After constructing the `Inner` object, its member function `fun()` is called.

That's about it. The function `Outer::fun()` terminates, and the destructor of the `Inner` object is called. Then the program terminates and the destructor of the `Outer` object is called. Here is the basic program:

```
#include <iostream>

class Inner
{
public:
    Inner();
    ~Inner();
    void fun();
};

class Outer
{
public:
    Outer();
```

```

        ~Outer();
        void fun();
    private:
};

Inner::Inner()
{
    cout << "Inner constructor\n";
}

Inner::~Inner()
{
    cout << "Inner destructor\n";
}

void Inner::fun()
{
    cout << "Inner fun\n";
}

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();
}

/*
    Generated output:
Outer constructor
Inner constructor
Outer fun
Inner fun
Inner destructor

```



```

    Outer destructor
    */

```

After compiling and running, the program's output is completely as expected, and it is exactly what we want: 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 to take place in the `Inner::fun()` function, which is supposedly handled somewhere at the end of the function `main()`. We'll consider two variants. The first variant will try to handle this situation using `setjmp()` and `longjmp()`; the second variant will try to handle this situation using C++'s exception mechanism.

### 8.2.1 No exceptions: 'setjmp()' and 'longjmp()'

In order to use `setjmp()` and `longjmp()` the basic program from section 8.2 is slightly modified to contain a variable `jmp_buf jmpBuf`. The function `Inner::fun()` now calls `longjmp`, simulating a disastrous event, to be handled at the end of the function `main()`. In `main()` we see the standard code defining the target location of the long jump, using the function `setjmp()`. A zero return value indicates the initialization of the `jmp_buf` variable, upon which the `Outer::fun()` function is called. This situation represents the 'normal flow'.

To complete the simulation, the return value of the program is zero *only* if the program is able to return from the function `Outer::fun()` normally. However, as we know, this won't happen: `Inner::fun()` calls `longjmp()`, returning to the `setjmp()` function, which (at this time) will *not* return a zero return value. Hence, after calling `Inner::fun()` from `Outer::fun()` the program proceeds beyond the `if`-statement in the `main()` function, and the program terminates with the return value 1. Now try to follow these steps by studying the following program source, modified after the basic program given in section 8.2:

```

#include <iostream>
#include <setjmp.h>
#include <stdlib.h>

class Inner
{
public:
    Inner();
    ~Inner();
    void fun();
};

class Outer
{
public:
    Outer();
    ~Outer();
    void fun();
};

jmp_buf
    jmpBuf;

```

```

Inner::Inner()
{
    cout << "Inner constructor\n";
}

void Inner::fun()
{
    cout << "Inner fun()\n";
    longjmp(jmpBuf, 0);
}

Inner::~Inner()
{
    cout << "Inner destructor\n";
}

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))
    {
        out.fun();
        return 0;
    }

    return 1;
}
/*
    Generated output:
Outer constructor
Inner constructor
Outer fun
Inner fun()
Outer destructor

```

```
*/
```

From the program generated by this program it is clear that the destructor of the class `Inner` is not executed. This is a direct result of the non-local characteristic of the call to `longjmp()`: processing proceeds immediately from the `longjmp()` call in the member function `Inner::fun()` to the function `setjmp()` in `main()`. There, its return value is zero, so the program terminates with return value 1. What is important here is that the call to the destructor `Inner::~Inner()`, waiting to be executed at the end of `Outer::fun()`, is never reached.

Since this example shows that the destructors of objects can easily be skipped when `longjmp()` and `setjmp()` are used, it's probably best to avoid these function completely in C++ programs.

### 8.2.2 Exceptions: the preferred alternative

In C++ *exceptions* are the best alternative to `setjmp()` and `longjmp()`. In this section an example using exceptions is presented. Again, the program is derived from the basic program, given in section 8.2:

```
#include <iostream>

class Inner
{
public:
    Inner();
    ~Inner();
    void fun();
};

class Outer
{
public:
    Outer();
    ~Outer();
    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";
}
```

```

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;
    try
    {
        out.fun();
    }
    catch (...)
    {}
}
/*
    Generated output:
Outer constructor
Inner constructor
Outer fun
Inner fun
Inner destructor
Outer destructor
*/

```

In this program an *exception* is thrown, where a `longjmp()` was used in the program in section 8.2.1. The comparable construct for the `setjmp()` call in that program is represented here by the `try` and `catch` blocks. The `try` block surrounds statements (including function calls) in which exceptions are thrown, the `catch` block may contain statements to be executed just after throwing an exception.

So, comparably to the example given in section 8.2.1, the execution of function `Inner::fun()` terminates, albeit with an exception, rather than a `longjmp()`. The exception is caught in `main()`, and the program terminates. When the output from the current program is inspected, we notice that the destructor of the `Inner` object, created in `Outer::fun()` is now correctly called. Also notice that the execution of the function `Inner::fun()` really terminates at the `throw` statement: the insertion of the text into `cout`, just beyond the `throw` statement, doesn't take place.

Hopefully we now have raised your appetite for exceptions by showing that

- Exceptions provide a means to break out of the normal flow control without having to use a

cascade of `return`-statements, and without the need to terminate the program.

- Exceptions do not disrupt the activation of destructors, and are therefore strongly preferred over the use of `setjmp()` and `longjmp()`.

## 8.3 Throwing exceptions

Exceptions may be generated in a `throw` statement. The `throw` keyword is followed by an expression, which results in a value of a certain type. For example:

```
throw "Hello world";      // throws a char *
throw 18;                 // throws an int
throw string("hello");    // throws a string
```

Objects defined locally in functions are automatically destroyed once exceptions are thrown within these functions. However, if the object itself is thrown, the exception catcher receives a copy of the thrown object. This copy is constructed just before the local object is destroyed.

The next example illustrates this point. Within the function `Object::fun()` a local `Object toThrow` is created, which is thereupon thrown as an exception. The exception is caught outside of `Object::fun()`, in `main()`. At this point the thrown object doesn't actually exist anymore, Let's first take a look at the sourcetext:

```
#include <iostream>
#include <string>

class Object
{
public:
    Object(string name)
    :
        name(name)
    {
        cout << "Object constructor of " << name << "\n";
    }
    Object(Object const &other)
    :
        name(other.name + " (copy)")
    {
        cout << "Copy constructor for " << name << "\n";
    }
    ~Object()
    {
        cout << "Object destructor of " << name << "\n";
    }
    void fun()
    {
        Object
            toThrow("'local object'");

        cout << "Object fun() of " << name << "\n";
        throw toThrow;
    }
};
```

```

    }
    void hello()
    {
        cout << "Hello by " << name << "\n";
    }
private:
    string
        name;
};

int main()
{
    Object
        out("'main object'");

    try
    {
        out.fun();
    }
    catch (Object o)
    {
        cout << "Caught exception\n";
        o.hello();
    }
}
/*

```

```

    Generated output:
Object constructor of 'main object'
Object constructor of 'local object'
Object fun() of 'main object'
Copy constructor for 'local object' (copy)
Object destructor of 'local object'
Copy constructor for 'local object' (copy) (copy)
Caught exception
Hello by 'local object' (copy) (copy)
Object destructor of 'local object' (copy) (copy)
Object destructor of 'local object' (copy)
Object destructor of 'main object'
*/

```

The class `Object` defines some simple constructors and members. The copy constructor is special in that it adds the text " (copy)" to the received name, to allow us to monitor the construction and destruction of objects more closely. The member function `Object::fun()` generates the exception, and throws its locally defined object. Just before the exception the following output is generated by the program:

```

Object constructor of 'main object'
Object constructor of 'local object'
Object fun() of 'main object'

```

Now the exception is generated, resulting in the next line of output:

```

Copy constructor for 'local object' (copy)

```

The `throw` clause receives the local object, and treats it as a value argument: it creates a copy of the local object. Following this, the exception is processed: the local object is destroyed, and the catcher catches an `Object`, again a value parameter. Hence, another copy is created. Therefore, we see the following lines:

```
Object destructor of 'local object'
Copy constructor for 'local object' (copy) (copy)
```

Now we are inside the catcher, who displays its message:

```
Caught exception
```

followed by the calling of the `hello()` member of the received object. This member also shows us that we received a *copy of the copy of the local object* of the `Object::fun()` member function:

```
Hello by 'local object' (copy) (copy)
```

Finally the program terminates, and its still living objects are now destroyed in their reversed order of creation:

```
Object destructor of 'local object' (copy) (copy)
Object destructor of 'local object' (copy)
Object destructor of 'main object'
```

If the catcher would have been implemented so as to receive a *reference* to an object (which you could do by using `catch (Object &o)`), then the double copy would have been avoided. In that case the output of the program would have been:

```
Object constructor of 'main object'
Object constructor of 'local object'
Object fun() of 'main object'
Copy constructor for 'local object' (copy)
Object destructor of 'local object'
Caught exception
Hello by 'local object' (copy)
Object destructor of 'local object' (copy)
Object destructor of 'main object'
```

This shows us that only a single copy of the local object has been used.

Of course, 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 dies once the exception is thrown, and the catcher receives a wild pointer. Bad news....

Summarizing, local objects are thrown as copied objects, pointers to local objects should not be thrown. However, it is possible to throw pointers or references to *dynamically* generated objects. In this case one must take care that the generated object is properly deleted when the generated exception is caught, to prevent a memory leak.

Exceptions are thrown in situations where a function can't continue its normal task anymore, although the program is still able to continue. Imagine a program which is an interactive calculator. The program continuously requests expressions, which are then evaluated. In this case the parsing of the expression may show syntactical errors; and the evaluation of the expression may result

in expressions which can't be evaluated, e.g., because of the expression resulting in a division by zero. Also, the calculator might allow the use of variables, and the user might refer to non-existing variables: plenty of reasons for exceptions to be thrown, but no overwhelming reason to terminate the program. In the program, the following code may be used, all throwing exceptions:

```
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";
```

The location of these `throw` statements is immaterial: they may be placed deeply nested within the program, or at a more superficial level. Furthermore, *functions* may be used to generate the expression which is then thrown. A function

```
char const *formatMessage(char const *fmt, ...);
```

would allow us to throw more specific messages, like

```
if (!lookup(variableName))
    throw formatMessage("Variable '%s' not defined", variableName);
```

### 8.3.1 The empty 'throw' statement

Situations may arise in which it is required to inspect a thrown exception. Depending on the nature of the received exception, the program may continue its normal operation, or a serious event took place, requiring a more drastic reaction by the program. In a server-client situation the client may enter requests to the server in a queue. Every request placed in the queue is normally answered by the server, telling the client that the request was successfully completed, or that some sort of error has occurred. Actually, 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's 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 code handling an exception the received exception is passed on to the next level that might be able to process that particular type of exception.

In our server-client situation a function

```
initialExceptionHandler(char *exception)
```

could be designed to do so. The received message is inspected. If it's a simple message it's processed, otherwise the exception is passed on to an outer level. The implementation of `initialExceptionHandler()` shows the empty `throw` statement:

```
void initialExceptionHandler(char *exception)
```



```

{
    if (plainMessage(exception))
        handleTheMessage(exception);
    else
        throw;
}

```

As we will see below (section 8.5), the empty `throw` statement passes on the exception received in a `catch`-block. Therefore, a function like `initialExceptionHandler()` can be used for a variety of thrown exceptions, as long as the argument used with `initialExceptionHandler()` is compatible with the nature of the received exception.

Does this sound intriguing? Then try to follow the next example, which jumps slightly ahead to the topics covered in chapter 14. The next example may be skipped, though, without loss of continuity.

We can now state that a basic exception handling class can be constructed from which specific exceptions are derived. Suppose we have a class `Exception`, containing a member function `ExceptionType Exception::severity()`. This member function tells us (little wonder!) the severity of a thrown exception. It might be `Message`, `Warning`, `Mistake`, `Error` or `Fatal`. Furthermore, depending on the severity, a thrown exception may contain less or more information, somehow processed by a function `process()`. In addition to this, all exceptions have a plain-text producing member function, e.g., `toString()`, telling us a bit more about the nature of the generated exception. Using polymorphism, `process()` can be made to behave differently, depending on the nature of a thrown exception, when called through a basic `Exception` pointer or reference.

In this case, a program may throw all these five types of exceptions. Let's assume that the `Message` and `Warning` exceptions are processable by our `initialExceptionHandler()`. Then its code would become:

```

void initialExceptionHandler(Exception const *e)
{
    cout << e->toString() << endl; // show the plain-text information

    if
    (
        e->severity() == ExceptionWarning
        ||
        e->severity() == ExceptionMessage
    )
    {
        e->process(); // Process a message or a warning
        delete e;
    }
    else
        throw; // Pass on other types of Exceptions
}

```

Due to polymorphism, `e->process()` will either process a `Message` or a `Warning`. Thrown exceptions are generated as follows:

```

throw new Message(<arguments>);
throw new Warning(<arguments>);
throw new Mistake(<arguments>);
throw new Error(<arguments>);
throw new Fatal(<arguments>);

```

All of these exceptions are processable by our `initialExceptionHandler()`, which may decide to pass exceptions upward for further processing or to process exceptions itself. The polymorphic exception class is developed further in section 14.7.

## 8.4 The try block

The `try`-block surrounds statements in which exceptions may be thrown. As we have seen, the actual `throw` statement can be placed everywhere, not necessarily within the `try`-block. It may, for example, be placed in a function which is called from within the `try`-block, either directly or indirectly.

The keyword `try` is followed by a set of curly braces, which acts like a standard C++ compound statement: multiple statements and variable definitions may be placed here.

It is possible (and very common) to create *levels* in which exceptions may be thrown. For example, code within the `main()` function is surrounded by a `try`-block, forming an outer level in which exceptions can be handled. Within the `try`-block in `main()`, functions are called which may also contain `try`-blocks, forming the next level in which exceptions may be placed. As we have seen (in section 8.3.1), exceptions thrown in inner level `try`-blocks may or may not be processed at that level. By placing an empty `throw` in an exception handler, the thrown exception is passed on to the next (outer) level.

If an exception is thrown outside of any `try`-block, then the default way to handle (uncaught) exceptions is used, which is normally to abort the program. Try to compile and run the following tiny program, and see what happens:

```
int main()
{
    throw "hello";
}
```

## 8.5 Catching exceptions

The `catch` block contains code that is executed when an exception is thrown. Since *expressions* are thrown, the `catch`-block must know what kind of exceptions it should be able to handle. Therefore, the keyword `catch` is followed by a parameter list having one parameter, which is the type of the exception-expression that is handled by the `catch` block. So, an exception handler for `char const *` exceptions will have the following form:

```
catch (char const *message)
{
    // code to handle the message
}
```

Earlier (section 8.3) we've seen that such a message doesn't have to be thrown as a static string. It's also possible for a function to return a string, which is then thrown as an exception. However, if such a function creates the string that is thrown as an exception *dynamically*, the exception handler will normally have to delete the allocated memory to prevent a memory leak.

Generally, close attention must be paid to the nature of the parameter of the exception handler, to make sure that exception: dynamically generated dynamically generated exceptions are deleted

once the handler has processed them. Of course, when an exception is passed on to an outer level exception handler, the received exception should *not* be deleted by the inner level handler.

Different kinds of exceptions may be thrown: `char *s`, ints, pointers or references to objects, etc.: all these different types may be used in throwing and catching exceptions. So, various types of exceptions may come out of a `try`-block. In order to catch all expressions that may emerge from a `try`-block, multiple exception handlers (i.e., `catch`-blocks) may follow the `try`-block.

The *order* in which the exception handlers are placed 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. So only one exception handler following a `try`-block will be executed. Consequently, exception handlers should be placed from the ones having the most specific parameters to the ones having more general parameters. For example, if exception handlers are defined for `char *s` and `void *s` (i.e., any old pointer) then the exception handler for the former exception type should be placed before the exception handler for the latter type:

```
try
{
    // throws all kinds of pointers
}
catch (char const *message)
{
    // processing thrown char pointers
}
catch (void *whatever)
{
    // processing all other thrown pointers
}
```

As an alternative to constructing different types of exception handlers for different types of exceptions, it is of course also possible to design a specific class whose objects contain information about the reason for the exception. Such an approach was mentioned earlier, in section 8.3.1. Using this approach, there's only one handler required, since we *know* we won't throw other types of exceptions:

```
try
{
    // code throws only Exception pointers
}
catch (Exception *e)
{
    e->process();
    delete e;
}
```

The use of the `delete e` statement in the above code is an indication that the `Exception` object was created dynamically.

When the code of an exception handler that is placed beyond a `try`-block has been processed, the execution of the program continues beyond the last exception handler following that `try`-block (unless the handler uses `return`, `throw` or `exit()` to leave the function prematurely). Therefore, we distinguish the following cases:

- If no exception was thrown within the `try`-block no exception handler is activated, and the

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 an other level contains an appropriate exception handler, the program's default exception handler is called, usually aborting the program.
- If an exception was thrown within the `try`-block and an appropriate exception handler is available, then that the code of that exception handler is executed. Following the execution of the code of the exception handler, the execution of the program continues at the first statement beyond the last `catch`-block.

In all cases a `throw`-statement will result in ignoring all remaining statements of the `try`-block in which the exception was thrown. However, destructors of objects defined locally in the `try`-block are called, and they are called before any exception handler's code is executed.

The actual computation or construction of the exception may be performed in various degrees of sophistication. Several possibilities are to use a plain `new` operator; to use static member functions of a class; to return a pointer to an object; or to use objects of classes derived from a class, possibly involving polymorphism.

### 8.5.1 The default catcher

In cases where different types of exceptions can be thrown, only a limited set of handlers may be required at a certain level of the program. Exceptions whose types belong to that limited set are processed, all other exceptions are treated differently, i.e., they are passed on to an outer level of exception handling.

An intermediate kind of exception handling may be implemented using the default exception handler, which should (due to the hierarchical nature of exception catchers, discussed in section 8.5) be placed beyond all other, more specific exception handlers. In this case, the current level of exception handling may do some processing by default, but will then, using the empty `throw` statement (see section 8.3.1), pass the thrown exception on to an outer level. Here is an example showing the use of a default exception handler:

```
#include <iostream>

int main()
{
    try
    {
        try
        {
            throw 12.25;    // no specific handler for doubles
        }
        catch (char const *message)
        {
            cout << "Inner level: caught char const *\n";
        }
        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 still knows the double: " << d << endl;
    }
}
/*
    Generated output:
    Inner level: generic handling of exceptions
    Outer level still knows the double: 12.25
*/

```

From the generated output we may conclude that an empty `throw` statement throws the received exception to the next (outer) level of exception catchers, keeping the type and value of the exception: basic or generic exception handling can thus be accomplished at an inner level, specific handling, based on the type of the thrown expression, can then continue at an outer level.

## 8.6 Declaring exception throwers

Functions that are defined elsewhere may be linked to code that use these functions. Such functions are normally declared in header files, either as stand alone functions or as member functions of a class.

These external functions may of course throw exceptions. The declaration of such functions may contain a *function throw list*, in which the types of the exceptions that can be thrown by the function are specified. For example, a function that may throw `'char *'` and `'int'` exceptions can be declared as

```
void exceptionThrower() throw(char *, int);
```

A function throw list may appear after the function header (including a possible `const` specifier), and, noting that the throw list may be empty, it has the following generic form: `throw([type1 [, type2, type3, ...]])`

If a function *doesn't* throw exceptions an empty function throw list may be used. E.g.,

```
void noExceptions() throw ();
```

In this case the function definition must contain the empty function throw list too.

A function for which a function throw list is specified may not throw other types of exceptions. A *run-time error* occurs if it tries to throw other types of exceptions than mentioned in the function throw list.

If a function throw list is specified in the *declaration*, it must also be given in the *definition* of the function. For example, using declaration *and* definition in the next example:

```
#include <iostream>
```

```

void charPintThrower() throw(char const *, int);    // declarations

class Thrower
{
    public:
        void intThrower(int) const throw(int);
};

void Thrower::intThrower(int x) const throw(int)    // definitions
{
    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()
{
    try
    {
        charPintThrower();
    }
    catch (char const *message)
    {
        cerr << "Text exception: " << message << endl;
    }
    catch (int value)
    {
        cerr << "Int exception: " << value << endl;
    }
    try
    {
        cerr << "Up to the run-time error\n";
        runTimeError();
    }
    catch(...)
    {
        cerr << "not reached\n";
    }
}

```

```
}
```

In the function `charPintThrower()` the `throw` statement clearly throws a `char const *`. However, since `intThrower()` may throw an `int` exception, the function throw list of `charPintThrower()` must *also* contain `int`.

If the function throw list is not used, the function may either throw exceptions (of any kind) or not throw exceptions at all. Without a function throw list the responsibility of providing the correct handlers is in the hands of the designer of the program.

## 8.7 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::exceptions()`: this member returns the state flags for which the stream will throw exceptions,
- `ios::exceptions(state)`: this member will throw an exception when state `state` is observed.

The exception that is thrown is an object of the class `ios::failure`, derived from `ios::exception`. A `failure` object can be constructed with a `string const &message`, which can be retrieved using the virtual `char const *what() const` member.

Exceptions should be used for truly exceptional situations. Therefore, we think it is questionable to have stream objects throw exceptions for rather standard situations like EOF. Considering exceptions when input errors occur might be defensible, for example when input errors should not occur and imply a corrupted file. But here we think aborting the program with an appropriate error message usually is a more appropriate line of action. In any case, here is an example showing the use of exceptions in an interactive program, expecting numbers:

```
#include <iostream>
#include <string>

using namespace::std;

int main()
{
    cin.exceptions(ios_base::failbit);

    while (true)
    {
        try
        {
            cout << "enter a number: ";

            int
                value;

            cin >> value;
```

```

        cout << "you entered " << value << endl;
    }
    catch (ios_base::failure const &problem)
    {
        cout << problem.what() << endl;
        cin.clear();
        string
            s;
        getline(cin, s);
    }
}
}

```



## Chapter 9

# More Operator Overloading

Having covered the overloaded assignment operator in chapter 7, and having shown several examples of other overloaded operators as well (i.e., the insertion and extraction operators in chapters 3 and 5), we will now take a look at several other interesting examples of operator overloading.

### 9.1 Overloading ‘operator[]()’

As our next example of operator overloading, we present a class which is meant to operate on an array of ints. Indexing the array elements occurs with the standard array operator [], but additionally the class checks for boundary overflow. Furthermore, the index operator (operator[]()) is interesting in that it both *produces* a value and *accepts* a value, when used, respectively, as a *right-hand value (rvalue)* and a *left-hand value (lvalue)* in expressions. An example showing the use of the class is:

```
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] << endl;
}
```

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 will also produce a run-time error as the non-existing value `x[20]` is addressed. The `IntArray` class interface is:

```
class IntArray
{
public:
    IntArray(unsigned size = 1);
```

```

IntArray(IntArray const &other);
~IntArray();
IntArray const &operator=(IntArray const &other);

// overloaded index operators:
int &operator[](unsigned index);    // first
int operator[](unsigned index) const; // second
private:
void boundary(unsigned index) const;
void copy(IntArray const &other);
int
    *data;
unsigned
    size;
};

```

This class has the following characteristics:

- The class has a constructor with a default `int` argument, 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 of them ?

The first overloaded index operator allows us to reach and modify the elements of non-constant `IntArray` objects. This overloaded operator has as its prototype a function that returns a *reference* to an `int`. This allows us to use expressions like `x[10]` as *rvalues* and as *lvalues*.

We can therefore use the same function to retrieve and to assign values. Furthermore note that the return value of the overloaded array operator is *not* an `int const &`, but rather an `int &`. In this situation we don't use the `const`, as we must be able to change the element we want to access, if the operator is used as an *lvalue*.

However, 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 a *const* object, which cannot be modified. The compiler detects this and will refuse to compile this object definition if only the first overloaded index operator is available. Hence the second overloaded index operator. Here the return-value is an `int`, rather than an `int &`, and the member-function itself is a *const* member function. This second form of the overloaded index operator is not used with *non-const* objects, but it's used with *const* objects. It can only be used for value-retrieval, not for value-assignment, but that is precisely what we want with *const* objects.

Furthermore note that, since the values stored in the `IntArray` are primitive values of type `int`, using value return types is ok here. 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. An `int const &` could have been used in the class `IntArray` as well. The second overloaded index operator would then get the following prototype:

```
int const & IntArray::operator[](int index) const
```

- As there is only one pointer data member, the destruction of the memory allocated by the object is a simple `delete` data. Therefore, our standard `destroy()` function was not used.
- As the elements of data are `ints`, no `delete[]` is needed.

Now, the implementation of the members are:

```
#include "intarray.h"
#include <iostream>

IntArray::IntArray(unsigned sz)
:
    size(sz)
{
    if (size < 1)
    {
        cerr << "IntArray: size of array must be >= 1\n";
        exit(1);
    }
    data = new int [size];
}

IntArray::IntArray(IntArray const &other)
{
    copy(other);
}

IntArray::~IntArray()
{
    delete data;
}

IntArray const &IntArray::operator=(IntArray const &other)
{
    if (this != &other)
    {
        delete data;
        copy(other);
    }
    return *this;
}

void IntArray::copy(IntArray const &other)
{
    size = other.size;
    data = new int [size];
    memcpy(data, other.data, size * sizeof(int));
}

int &IntArray::operator[](unsigned index)
{
    boundary(index);
    return data[index];
}

int IntArray::operator[](unsigned index) const
{
    boundary(index);
    return data[index];
}
```

```

}

void IntArray::boundary(unsigned index) const
{
    if (index >= size)
    {
        cerr << "IntArray: boundary overflow, index = " <<
            index << ", should range from 0 to " << size - 1 << endl;
        exit(1);
    }
}
}

```

## 9.2 overloading the insertion and extraction operators

This section describes how a class can be adapted in such a way that it can be used with the C++ streams `cout` and `cerr` and the insertion operator `operator<<()`. Adapting a class in such a way that the `istream`'s extraction operator (`operator>>()`) can be used, occurs in a similar way and is simply shown in an example.

The implementation of an overloaded `operator<<()` in the context of `cout` or `cerr` involves their class, which is `ostream`. This class is declared in the header file `ostream` and defines only overloaded operator functions for 'basic' types, such as, `int`, `char *`, etc.. The purpose of this section is to show how an insertion operator can be overloaded in such a way that an object of any class, say `Person` (see chapter 7), can be inserted into an `ostream`. After making available such an overloaded operator, the following will be possible:

```

Person
    kr("Kernighan and Ritchie", "unknown", "unknown");

cout << "Name, address and phone number of Person kr:\n" << kr << endl;

```

The statement `cout << kr` involves `operator<<()`. This member function has two operands: an `ostream &` and a `Person &`. The proposed action is defined in an overloaded global operator `operator<<()` expecting two arguments:

```

// assume declared in 'person.h'
ostream &operator<<(ostream &, Person const &);

// define in some source file
ostream &operator<<(ostream &stream, Person const &pers)
{
    return
        stream <<
            "Name:      " << pers.getName() <<
            "Address:  " << pers.getAddress() <<
            "Phone:    " << pers.getPhone();
}

```

Note the following characteristics of `operator<<()`:

- The function returns a reference to an `ostream` object, to enable 'chaining' of the insertion operator.

- The two operands of `operator<<()` act as arguments of the overloaded function. In the earlier example, the parameter `stream` is initialized by `cout`, the parameter `pers` is initialized by `kr`.

In order to overload the extraction operator for, e.g., the `Person` class, members are needed to modify the private data members. Such *modifiers* are normally included in the class interface. For the `Person` class, the following members should be added to the class interface:

```
void setName(char const *name);
void setAddress(char const *address);
void setPhone(char const *phone);
```

The implementation of these members could be straightforward: 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.,

```
void Person::setAddress(char const *address)
{
    delete address;
    address = strdupnew(address);
}
```

A more elaborate function could also check the reasonableness of the new address. This elaboration, however, is not further pursued here. Instead, let's have a look at the final overloaded extraction operator (`operator>>()`). A simple implementation is:

```
istream &operator>>(istream &str, Person &p)
{
    string
        name,
        address,
        phone;

    if (str >> name >> address >> phone)    // extract three strings
    {
        p.setName(name.c_str());
        p.setAddress(address.c_str());
        p.setPhon(phone.c_str());
    }
    return str;
}
```

Note the stepwise approach that is followed with the extraction operator: first the required information is extracted, using available extraction operators (like a `string`-extraction), then, if that succeeds, *modifier* members are used to modify the data members of the object to be extracted. Finally, the stream object itself is returned as a reference.

## 9.3 Conversion operators

A class may be constructed around a basic type. E.g., the class `string` was 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:

```

class String
{
    public:
        String();
        String(char const *arg);
        ~String();
        String(String const &other);
        String const &operator=(String const &rvalue);
        String const &operator=(char const *rvalue);
    private:
        char
            *string;
};

```

Objects from 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 *`<sup>1</sup>.

Usually, in classes that are less directly linked to their data than this `String` class, there will be an *accessor member function*, like `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, we would have the following interface for `StringArray`:

```

class StringArray
{
    public:
        StringArray(unsigned size);
        StringArray(StringArray const &other);
        StringArray const &operator=(StringArray const &rvalue);
        ~StringArray();

        String &operator[](unsigned index);
    private:
        String
            *store;
        unsigned
            n;
};

```

Using the `StringArray::operator []`, assignments between the `String` elements can simply be realized:

```

StringArray
    sa(10);

sa[4] = sa[3]; // String to String assignment

```

It is also possible to assign a `char const *` to an element of `sa`:

```

sa[3] = "hello world";

```

---

<sup>1</sup> Note that the assingment from a `char const *` also includes the null-pointer. An assignment like `stringObject = 0` is perfectly in order.

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]` can receive 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]`? We try the following code:

```
char const
    *cp = sa[3];
```

This, however, won't work: 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 9.10). Furthermore, *casting* won't work: the compiler doesn't know how to cast a `String` to a `char const *`. How to proceed from here?

The naive solution is to resort to the accessor member function `c_str()`:

```
cp = sa[3].c_str()
```

That solution would work, but it looks so 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. Using a conversion operator a `String` object may be interpreted as a `char const *`, which can then be assigned to another `char const *`. Conversion operators can be implemented for all types for which a conversion is needed.

In the current example, the class `String` would need a conversion operator for a `char const *`. In class interfaces, the general form of a conversion operator is:

```
operator <type>();
```

In our `String` class, this would become:

```
operator char const *();
```

The implementation of the conversion operator is straightforward:

```
String::operator char const *()
{
    return (string);
}
```

Notes:

- There is *no* mentioning of a return type. The conversion operator returns a value of the type mentioned after the operator keyword.

- In certain situations the compiler needs a hand to disambiguate our intentions. In a statement like

```
cout.form("%s", sa[3])
```

the compiler is confused: are we going to pass a `String &` or a `char const *` to the `form()` member function? To help the compiler, we supply an `static_cast`:

```
cout.form("%s", static_cast<char const *>(sa[3]));
```

One might wonder what will happen if an object for which, e.g., a `string` conversion operator is defined is inserted into, e.g., an `ostream` object, into which `string` objects can be inserted. In this case, the compiler will not look for appropriate conversion operators (like `operator string()`), but will report an error. For example, the following example procedure a compilation error:

```
#include <iostream>
#include <string>

class NoInsertion
{
public:
    operator string() const;
};

int main()
{
    NoInsertion
        object;

    cout << object << endl;
}
```

The problem is caused by the fact that the compiler notices an insertion, applied to an object. It will now look for an appropriate overloaded version of the insertion operator. As it can't find one, it reports a compilation error, instead of performing a two-stage insertion: first using the `operator string()` insertion, followed by the insertion of that `string` into the `ostream` object. Conversion operators are used when the compiler is given no choice: an assignment of a `NoInsertion` object to a `string` object is such a situation. The problem of how to insert an object into, e.g., an `ostream` is simply solved: by defining an appropriate overloaded insertion operator, rather than by resorting to a conversion operator.

## 9.4 The 'explicit' keyword

Conversions are not only performed by conversion operators, but also by constructors having one parameter (or multiple parameters, having default parameter values beyond the first parameter).

Consider the class `Person` introduced in chapter 7. This class has a constructor

```
Person(char const *name, char const *address, char const *phone)
```



This constructor could be given default parameter values:

```
Person(char const *name, char const *address = "<unknown>",
        char const *phone = "<unknown>");
```

In several situations this constructor might be used intentionally, possibly providing the default <unknown> texts for the address and phone numbers. For example:

```
Person
    frank("Frank", "Room 352", "050 363 3688");
```

Also, functions might use `Person` objects as parameters, e.g., the following member in a fictitious class `PersonData` could be available:

```
PersonData &PersonData::operator+=(Person const &person);
```

Now, combining the above two pieces of code, we might, do something like

```
PersonData
    dbase;

dbase += frank;    // add frank to the database
```

So far, so good. However, since the `Person` constructor can also be used as a conversion operator, it is *also* possible to do:

```
dbase += "karel";
```

Here, the `char const *text` 'karel' is converted to an (anonymous) `Person` object using the former `Person` constructor: the second and third parameters use their default values. Here, and *implicit conversion* is performed from a `char const *` to a `Person` object, which might not be what the programmer had in mind when the class `Person` was constructed.

As another example, consider the situation where a class representing a container is constructed. Let's assume that the initial construction of objects of this class is rather complex and time-consuming, but *expanding* an object so that it can accomodate more elements is even more time-consuming. Such a situation might arise when a hash-table is initially constructed to contain `n` elements: that's ok as long as the table is not full, but when the table must be expanded, all its elements normally must be rehashed according to the new table size.

Such a class could (partially) be defined as follows:

```
class HashTable
{
public:
    HashTable(unsigned n); // n: initial table size
    unsigned size();       // returns current # of elements
                           // add new key and value

    HashTa
    void add(string const &key, string const &value);
};
```

Now consider the following implementation of `add()`:

```
void HashTable::add(string const &key, string const &value)
{
    if (size() > n * 0.75) // table gets rather full
        ht = size() * 2;  // Oops: not what we want!

    // etc.
}
```

In the first line of the body of `add()` the programmer first determines how full the hashtable currently is: if it's more than three quarter full, then the intention is to double the size of the hashtable. This eventually succeeds, but the cost is way too high: accidentally the programmer assigns an unsigned value, intending to tell the hashtable what its new size should be. What happens next comes as a bit of a surprise:

- The compiler notices that no `operator=(unsigned newsize)` is available for `HashTable`.
- There is, however, a constructor accepting an unsigned, *and* there is a copy constructor.
- So, the rvalue of the assignment is used to (implicitly) construct an (empty) `HashTable` that can accomodate `2 * size()` elements.
- Next, the just constructed empty `HashTable` is now assigned to the current `HashTable`, thus *removing all hitherto stored elements from the current HashTable*.

If an implicit use of a constructor is not appropriate (or dangerous), it can be prevented by using the `explicit` modifier with the constructor. Constructors using the `explicit` modifier can only be used for the explicit construction of objects, and cannot be used as implicit type convertors anymore. For example, to prevent the implicit conversion from `char const *` to `Person` the class interface of the class `Person` must contain the constructor

```
explicit Person(char const *name, char const *address = "<unknown>",
               char const *phone = "<unknown>");
```

Note, that it is still possible to add 'karel' to the `dbase` object, but it can't be realized through an implicit conversion anymore. To add 'karel' to the database, the `Person` constructor must be explicitly called, creating an anonymous object:

```
dbase += Person("karel");
```

Similarly, the assignment `ht = size() * 2` is now caught by the compiler, complaining with a message like *no match for 'HashTable& = unsigned int' operator*.

## 9.5 Overloading increment and decrement

Overloading the increment operator (`operator++()`) and decrement operator (`operator--()`) creates a little problem: there are two version 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 *rvalue*, so we have an expression that has a fixed value: the post-incremented variable itself disappears from view. Used as *prefix operator*,

the variable is incremented, and its value is returned as *lvalue*, so it can be altered immediately again. 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 unsigned value type. The class could have the following (partially shown) interface:

```
class Unsigned
{
    public:
        Unsigned();
        Unsigned(unsigned init);
        Unsigned &operator++();
    private:
        unsigned
            value;
}
```

This defines the *prefix* overloaded increment operator. An *lvalue* is returned, as we can deduce from the return type, which is `Unsigned &`.

The *implementation* of the above function could be:

```
Unsigned &Unsigned::operator++()
{
    ++value;
    return *this;
}
```

In order to define the *postfix* operator, an overloaded version of the operator is defined, expecting an `int` argument. This might be considered a *kludge*, or a consequent application of the notion of overloaded functions. Whatever your opinion in this matter, we can conclude the following:

- 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 the value the object has at the point the overloaded operator is called.

To add the postfix increment operator to the `Unsigned` wrapper class, add the following line to the class interface:

```
Unsigned &operator++(int);
```

The *implementation* of the postfix increment operator should be like this:

```
Unsigned Unsigned::operator++(int)
{
    return value++;
}
```

The simplicity of this implementation is *deceiving*, however. Note that:

- `value` is used with a postfix increment in the `return` expression. Therefore, the value of the `return` expression is `value`'s value, before it is incremented; which is correct.

- The return value of the function is an `Unsigned` value. This *anonymous object* is implicitly initialized by the value of `value`, so there is a hidden constructor call here.
- Anonymous objects are always `const` objects, so, indeed, the return value of the postfix increment operator is an *rvalue*.
- The parameter is not used. It is only part of the implementation to *disambiguate* the prefix- and postfix operators in implementations and declarations.

When the object has a more complex data organization, the use of a copy constructor might be indicated. For instance, assume we want to implement the postfix increment operator in the class `PersonData`, mentioned in section 9.4. Presumably, the `PersonData` class contains a complex inner data organization. If the `PersonData` class would maintain a pointer `Person *current` to the `Person` object that is currently selected, then the postfix increment operator for the class `PersonData` could be implemented as follows:

```
PersonData PersonData::operator++(int)
{
    PersonData
        tmp(*this);

    incrementCurrent();    // increment 'current', somehow.
    return tmp;
}
```

A matter of concern here could be that this operation actually requires *two* calls to the copy constructor: first to keep the current state, then to copy the `tmp` object to the (anonymous) return value. In some cases this double call of the copy constructor might be avoidable, by defining a specialized constructor. E.g.,

```
PersonData PersonData::operator++(int)
{
    return PersonData(*this, incrementCurrent());
}
```

Here, `incrementCurrent()` is supposed to return the information which allows the constructor to set its `current` data member to the pre-increment value, while at the same time incrementing `current` of the actual `PersonData` object. The above constructor would have to:

- initialize its data members by copying the values of the data members of the `this` object.
- reassign `current` based on the return value of its second parameter, which could be, e.g., an index.

At the same time, `incrementCurrent()` would have incremented `current` of the actual `PersonData` object.

The general rule is that double calls of the copy constructor can be avoided if a specialized constructor can be defined which initializes an object to the pre-increment state of the current object. The current object itself has its necessary data members incremented by a function, whose return value is passed as argument to the constructor, thereby informing the constructor of the pre-incremented state of the involved data members. The postfix increment operator will then return the thus constructed (anonymous) object, and no copy constructor is ever called.

Finally it is noted that the call of the increment or decrement operator using its overloaded function name might require us to provide a (any) `int` argument to inform the compiler that we want the postfix increment function. E.g.,

```

PersonData
    p;

p = other.operator++();    // incrementing 'other', then assigning 'p'
p = other.operator++(0);   // assigning 'p', then incrementing 'other'

```

## 9.6 Overloading 'operator new(size\_t)'

If the operator `new` is overloaded, it must have a `void *` return type, and at least an argument of type `size_t`. The `size_t` type is defined in the header file `cstddef`, which must therefore be included when the operator `new` is overloaded.

It is also possible to define multiple versions of the operator `new`, as long as each version has its own unique set of arguments. The global `new` operator can still be used, through the `::`-operator. If a class `X` overloads the operator `new`, then the system-provided operator `new` is activated by

```
X *x = ::new X();
```

Overloading `new[]` is discussed in section 9.8. The following example shows an overloaded version of operator `new`:

```

#include <cstddef>

void *X::operator new(size_t sizeofX)
{
    void
        *p = new char[sizeofX];

    return memset(p, 0, sizeof(X));
}

```

Now, let's see what happens when operator `new` is overloaded for the class `X`. Assume that class is defined as follows<sup>2</sup>:

```

class X
{
    public:
        void *operator new(size_t sizeofX);

        int
            x,
            y;
};

```

---

<sup>2</sup>For the sake of simplicity we have violated the principle of encapsulation here. The principle of encapsulation, however, is immaterial to the discussion of the workings of the operator `new`.

Now, consider the following program fragment:

```
#include "x.h" // class X interface
#include <iostream>

int main()
{
    X
    *x = new X();

    cout << x->x << ", " << x->y << endl;
}
```

This small program produces the following output:

0, 0

At the call of `new X()`, our little program performed the following actions:

- First, `operator new` was called, which allocated and initialized a block of memory, the size of an `X` object.
- Next, a pointer to this block of memory was passed to the (default) `X()` constructor. Since no constructor was defined, the constructor itself didn't do anything at all.

Due to the initialization of the block of memory by `operator new` the allocated `x` object was already initialized to zeros when the constructor was called.

Non-static member functions are passed a (hidden) pointer to the object on which they should operate. This hidden pointer becomes the `this` pointer in non-static member functions. This procedure is also followed by the constructor. In the next pieces of pseudo C++ code, the pointer is made visible. In the first part an `X` object `x` is defined directly, in the second part of the example the (overloaded) `operator new` is used:

```
X::X(&x); // x's address is passed to the constructor

void // new allocates the memory for an X object
    *ptr = X::operator new();

X::X(ptr); // now let the constructor operate on the
           // memory returned by 'operator new'
```

Notice that in the pseudo C++ fragment the member functions were treated as static member function of the class `X`. Actually, `operator new` is a static member function of its class: it cannot reach data members of its object, since it's normally the task of the `operator new` first to create room for that object. It can do that by allocating enough memory, and by initializing the area as required. Next, the memory is passed to the constructor (as the `this` pointer) for further processing. The fact that an overloaded `operator new` is actually a static function, not requiring an object of its class, can be illustrated in the following (frowned upon in normal situations!) program fragment, which can be compiled without problems (assume class `X` has been defined and is available as before):

```
int main()
```

```

{
    X
    x;

    X::operator new(sizeof x);
}

```

The call to `X::operator new()` returns a `void *` to an initialized block of memory, the size of an `X` object.

The `operator new` can have multiple parameters. The first parameter is initialized by an implicit argument and is always the `size_t` parameter, other parameters are initialized by explicit arguments that are specified when `operator new` is used. For example:

```

class X
{
public:
    void *operator new(size_t p1, unsigned p2);
    void *operator new(size_t p1, char const *fmt, ...);
};

int main()
{
    X
    *p1 = new(12) X(),
    *p2 = new("%d %d", 12, 13) X(),
    *p3 = new("%d", 12) X();
}

```

The pointer `p1` is a pointer to an `X` object for which the memory has been allocated by the call to the first overloaded `operator new`, followed by the call of the constructor `X()` for that block of memory. The pointer `p2` is a pointer to an `X` object for which the memory has been allocated by the call to the second overloaded `operator new`, followed again by a call of the constructor `X()` for its block of memory. Notice that pointer `p3` also uses the second overloaded `operator new()`, as that overloaded operator accepts a variable number of arguments, the first of which is a `char const *`.

## 9.7 Overloading ‘operator delete(void \*)’

The `delete` operator may be overloaded too. The `operator delete` must have a `void *` argument, and an optional second argument of type `size_t`, which is the size in bytes of objects of the class for which the `operator delete` is overloaded. The return type of the overloaded `operator delete` is `void`.

Therefore, in a class the `operator delete` may be overloaded using the following prototype:

```
void operator delete(void *);
```

or

```
void operator delete(void *, size_t);
```

Overloading `delete[]` is discussed in section 9.8.

The 'home-made' operator `delete` is called after executing the destructor of the associated class. So, the statement

```
delete ptr;
```

with `ptr` being a pointer to an object of the class `X` for which the operator `delete` was overloaded, boils down to the following statements:

```
X::~X(ptr);    // call the destructor function itself

                // and do things with the memory pointed to by ptr
X::operator delete(ptr, sizeof(*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 activated using the `::` scope resolution operator. For example:

```
void X::operator delete(void *ptr)
{
    // any operation considered necessary, then:
    ::delete ptr;
}
```

## 9.8 Operators 'new[]' and 'delete[]'

In sections 7.1.1, 7.1.2 and 7.2.1 operator `new[]` and operator `delete[]` were introduced. Like operator `new` and operator `delete` the operators `new[]` and `delete[]` may be overloaded. Because 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 be followed:

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.

The default way these operators act is 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 an object, the destructor of its class is called.
- operator `new[]` is used to allocate a series of primitive values or objects. Note that if a series of objects is allocated, the class' default constructor is called to initialize each individual object.
- operator `delete[]` is used to delete the memory previously allocated by `new[]`. If objects were previously allocated, then the destructor will be called for each individual object. However, if *pointers to objects* were allocated, *no destructor is called*, as a pointer is considered a primitive type, and certainly not an object.



Operators `new[]` and `delete[]` may only be overloaded in classes. Consequently, when allocating primitive types or pointers to objects only the default line of action is followed: when arrays of pointers to objects are deleted, a memory leak occurs unless the objects to which the pointers point were deleted earlier.

In this section the mere syntax for overloading operators `new[]` and `delete[]` is presented. It is left as an exercise to the reader to make good use of these overloaded operators.

To overload operator `new[]` in a class `Object` the interface should contain the following lines, showing multiple forms of overloaded forms of operator `new[]`:

```
class Object
{
    public:
        void *operator new[](size_t size);
        void *operator new[](size_t index, unsigned extra);
};
```

The first form shows the basic form of operator `new[]`. It should return a `void *`, and defines at least a `size_t` parameter. When operator `new[]` is called, `size` contains the number of *bytes* that must be allocated for the required number of objects. These objects can be initialized by the *global* operator `new[]` using the form

```
::new Object [size / sizeof(Object)]
```

Alternatively, using

```
::new char [size]
```

the required (uninitialized) amount of memory can be allocated too. An example of an overloaded operator `new[]` member function, returning an array of `Object` objects all filled with 0-bytes, is:

```
void *Object::operator new[](size_t size)
{
    return memset(new char[size], 0, size);
}
```

Having constructed the overloaded operator `new[]`, it will be used automatically in statements like:

```
Object
    *op = new Object[12];
```

Operator `new[]` may be overloaded using extra parameters. The second form of the overloaded operator `new[]` shows such an extra unsigned parameter. The definition of such a function is standard, and could be:

```
void *Object::operator new[](size_t size, unsigned extra)
{
    unsigned
        n = size / sizeof(Object);
    Object
```

```

        *op = ::new Object[n];

    for (unsigned idx = 0; idx < n; idx++)
        op[idx].value = extra;          // assume a member 'value'

    return cp;
}

```

To use this overloaded operator, only the extra parameter must be provided. It is given in a parameter list just after the name of the operator itself:

```

Object
    *op = new(100) Object[12];

```

This results in an array of 12 Object objects, all having their value member set to 100.

Like operator new[] operator delete[] may be overloaded. To overload operator delete[] in a class Object the interface should contain the following lines, showing multiple forms of overloaded forms of operator delete[]:

```

class Object
{
    public:
        void operator delete[](void *p);
        void *operator delete[](void *p, size_t index);
        void *operator delete[](void *p, int extra, bool yes);
};

```

The first form shows the basic form of operator delete[]. Its parameter is initialized to the address of a block of memory previously allocated by Object::new[]. These objects can be deleted by the *global operator delete[]* using the form . However, the compiler expects ::delete[] to receive a pointer to Objects, so a type cast is necessary:

```

::delete[] reinterpret_cast<Object *>(p)

```

; An example of an overloaded operator delete[] is:

```

void Object::operator delete[](void *p)
{
    cout << "operator delete[] for Objects called\n";
    ::delete [] reinterpret_cast<Object *>(p);
}

```

Having constructed the overloaded operator delete[], it will be used automatically in statements like:

```

delete[] new Object[5];

```

Operator delete[] may be overloaded using extra parameters. However, if overloaded as

```

void *operator delete[](void *p, size_t size)

```

then `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 the first form should *not* be defined, to avoid ambiguity. An example of this form of `operator delete[]` is:

```
void Object::operator delete[](void *p, unsigned size)
{
    cout << "deleting " << size << " bytes\n";
    ::delete [] reinterpret_cast<Object *>(p);
}
```

If other parameters are defined, as in

```
void *operator delete[](void *p, int extra, bool yes)
```

an explicit argument list must be provided. With `delete[]`, the argument list is specified *behind* the brackets:

```
delete[](new Object[5], 100, false);
```

## 9.9 Function Objects

*Function Objects* are created by overloading the *function call operator* `operator()()`. By defining the function call operator an object may be used as a function, hence the term *function objects*.

Function objects play an important role in the *generic algorithms* and they can be used profitably as alternatives to using pointers to functions. The fact that they are important in the context of the generic algorithms constitutes some sort of a didactical dilemma: at this point it would have been nice if the generic algorithms would have been covered, but for the discussion of the generic algorithms knowledge of function objects is an advantage. This bootstrapping problem is solved in a well known way: by ignoring the dependency.

Function objects are objects for which the `operator()()` has been defined. Usually they are used in combination with generic algorithms, but they are also used in situations where otherwise pointers to functions would have been used. Another reason for using function objects is to support inline functions, which cannot be used in combination with pointers to functions.

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. A program fragment in which this function is used is, e.g.,

```
Person
    &target = targetPerson(), // determine the person we're looking for
    *pArray;
unsigned
    n = fillPerson(&pArray);

cout << "The target person is";

if (!lsearch(&target, pArray, &n, sizeof(Person), compareFunction))
    cout << " not";
```

```
cout << "found\n";
```

The function `targetPerson()` is called to determine the person we're looking for, and the function `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 the `lsearch()` function. It could be something like:

```
int compareFunction(Person const *p1, Person const *p2)
{
    return (*p1 != *p2);    // lsearch() wants 0 for equal objects
}
```

This, of course, assumes that the `operator!=()` has been overloaded in the class `Person`, as it is quite unlikely that a bitwise comparison will be appropriate here. But overloading `operator!=()` is no big deal, so let's assume that that operator is available as well.

With `lsearch()` (and friends, having parameters that are pointers to functions) an *inline* compare function cannot be used: as the address of the `compare()` function must be known to the `lsearch()` function. So, on the average  $n / 2$  times at *least* the following actions take place:

- The two arguments of the comparefunction are pushed on the stack;
- The final parameter of `lsearch()` is evaluated, producing the address of `compareFunction()`;
- The comparefunction is called;
- The address of the right-hand argument of the `Person::operator!=()` argument is pushed on the stack;
- The `operator!=()` function is evaluated;
- The argument of `Person::operator!=()` argument is popped off the stack;
- The two arguments of the comparefunction are popped off the stack.

When function objects are used a different picture emerges. Assume we have constructed a function `PersonSearch()`, having the following prototype (realize that this is not the preferred approach. Normally a generic algorithm will be preferred to a home-made function. But for now our `PersonSearch()` function is used to illustrate the use and implementation of a function object):

```
Person const *PersonSearch(Person *base, size_t nmemb,
                           Person const &target);
```

This function can be used as follows:

```
Person
    &target = targetPerson(),
    *pArray;
unsigned
    n = fillPerson(&pArray);

cout << "The target person is";
```

```

if (!PersonSearch(pArray, n, target))
    cout << " not";

cout << "found\n";

```

So far, nothing much has been altered. We've replaced the call to `lsearch()` with a call to another function: `PersonSearch()`. Now we show what happens inside `PersonSearch()`:

```

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;
}

```

The implementation shows a plain linear search. However, in the for-loop the expression `target(base[idx])` shows our target object used as a function object. Its implementation can be simple:

```

int Person::operator()(Person const &other) const
{
    return *this != other;
}

```

Note the somewhat peculiar syntax: `operator()()`. The first set of parentheses define the particular operator that is overloaded: the function call operator. The second set of parentheses define the parameters that are required for this function. `Operator()()` appears in the class header file as:

```

bool operator()(Person const &other) const;

```

Now, `Person::operator()()` is a simple function. It contains but one statement, so we could consider making it inline. Assuming that we do, then this is what happens when `operator()()` is called:

- The address of the right-hand argument of the `Person::operator!=(())` argument is pushed on the stack,
- The `operator!=(())` function is evaluated,
- The argument of `Person::operator!=(())` argument is popped off the stack,

Note that due to the fact that `operator()()` is an inline function, it is not actually called. Instead `operator!=(())` is called immediately. Also note that the required stack operations are fairly modest.

So, function objects may be defined inline. This is not possible for functions that are called indirectly (i.e., using pointers to functions). Therefore, even if the function object needs to do very little work it has to be defined as an ordinary function if it is going to be called via pointers. The overhead of performing the indirect call may not outweigh the advantage of the flexibility of calling functions indirectly. In these cases function objects that are defined as inline functions can result in an increase of efficiency of the program.

Finally, function objects may of course access their private data directly. A search algorithm where a compare function is used (as with `lsearch()`) the target and array elements are passed to the compare function using pointers, involving extra stack handling. When function objects are used, the target person doesn't vary within a single search task. Therefore, the target person could be passed to the constructor of the function object doing the comparison. This is in fact what happened in the expression `target(base[idx])`, where only one argument is passed to the `operator()()` member function of the target function object.

Actually, in the above example `operator()()` could have been avoided altogether in the above example. However, function objects play a central role in generic algorithms. In chapter 17 these generic algorithms are further discussed. Also in that chapter, the importance of function objects will be further emphasized when *predefined function objects* are discussed.

### 9.9.1 Constructing manipulators

In chapter 5 we saw constructions like `cout << hex << 13 << endl` 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 will set the field width of the next field to be written to the `ostream` object to 10. This manipulator is constructed as a function. The `w10` function will have to know about the `ostream` object in which the width must be set. By providing the function with a `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:

```
#include <iostream>
#include <iomanip>

ostream &w10(ostream &str)
{
    return str << 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.,

```
#include <iostream>
#include <iomanip>

extern ostream &w10(ostream &str);

int main()
{
    w10(cout) << 3 << " ships sailed to America" << endl;
    cout << "And " << w10 << 3 << " other ships sailed too." << endl;
}
```

The `w10` function can be used as a manipulator because the `ostream` class has an overloaded `operator<<` accepting a pointer to a function expecting an `ostream &` and returning an `ostream &`. Its definition is:

```
ostream& operator<<(ostream & (*func)(ostream &str))
{
    return ((*func)(*this));
}
```

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 `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:

```
cout << 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 unwanted side-effects. The following section introduces a way to implement manipulators requiring arguments without resorting to macros, but using anonymous objects.

### 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: in a sequence of `operator<<()` calls the compiler will call the functions first, and then use their return values in the insertion sequence. That will invalidate the ordering of the arguments passed to your `ii` operators.

So, one might consider constructing another overloaded `operator<<()` accepting the address of a function receiving not just the `ostream` reference, but a series of other arguments as well. The problem now is that it isn't clear how the function will receive its arguments: you can't just call it, since that produces the abovementioned problem, and you can't just pass its address in the insertion sequence, as you normally do with a manipulator....

However, there is a solution, based on the use of anonymous objects:

- First, a class is constructed, e.g. `Manip`, whose constructor expects the argument or multiple arguments we need to use.
- Furthermore, the class defines the `friend`

```
ostream &operator<< (ostream & ostr, Manip const &mm)
```

so a `Manip` object can be inserted into the `ostream`.

The class `Manip` could be, e.g.,

```
class Manip
{
```

```

friend ostream &operator<<(ostream &str, Manip const &x)
{
    return str << x.value;
}
public:
    Manip(int x)
    :
        value(x)
    {}
private:
    int
        value;
};

```

Now we're getting to where we want to be: by inserting anonymous `Manip` objects into an `ostream` the desired effect is reached: manipulators having (multiple) arguments. E.g.,

```

int main()
{
    cout << Manip(4) << " -- " <<
        Manip(5) << " ++ " <<
        Manip(6) << endl;
}
/*
    generated output:
    4 -- 5 ++ 6
*/

```

Note that in order to be able to insert an anonymous `MultiMap` object into the `ostream`, the `MultiMap`'s `operator<<()` friend *must* define a `Manip const &` parameter.

## 9.10 Overloadable Operators

The following operators can be overloaded:

+	-	*	/	%	~	&	
~	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	
+=	-=	*=	/=	%=	^=	&=	=
<<=	>>=	[]	()	->	->*	new	delete

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 9.3.



## Chapter 10

# Static data and functions

In the previous chapters we have shown examples of classes where each object of a class had its own set of `public` or `private` data. Each `public` or `private` function could access the members of objects of that class.

In some situations it may be desirable that one or more *common data fields* exist, which are accessible to *all* objects of the class. An example of such a situation is the name of the startup directory in a program which recursively scans the directory tree of a disk. A second example is a flag variable, which states whether some specific initialization has occurred: only the first object of the class would perform the necessary initialization and would set the flag to 'done'.

Such situations are analogous to C code, 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 declare the variable as a `static`: the variable name is then not known beyond the scope of the source file. This approach is quite valid, but doesn't stroke with our philosophy of one function per source file. Another C-solution is to give the variable in question an unusual name, e.g., `_6uldv8`, and then to hope that other program parts won't use this name by accident. Neither the first, nor the second C-like solution is elegant.

C++'s solution is to define `static` members: data and functions, common to all objects of a class and inaccessible outside of the class. These functions and data will be discussed in this chapter.

### 10.1 Static data

A data member of a class can be declared `static`; be it in the `public` or `private` part of the class definition. 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 separate object of the class.

Static data members are created when the program starts executing. Note, however, that they are always created as members of their classes.

Static data members which are declared `public` are like 'normal' global variables: they can be reached by *all code of the program* by simply using their names, together with their class names and the scope resolution operator. This is illustrated in the following example:

```
class Test
{
```

```

    public:
        static int
            public_int;
    private:
        static int
            private_int;
};

int main()
{
    Test::public_int = 145;    // ok

    Test::private_int = 12;    // wrong, don't touch
                               // the private parts

    return (0);
}

```

This code fragment is not suitable for consumption by a C++ compiler: it only illustrates the *interface*, and not the *implementation* of static data members, which is discussed next.

### 10.1.1 Private static data

To illustrate the use of a static data member which is a private variable in a class, consider the following example:

```

class Directory
{
    public:
        // constructors, destructors, etc. (not shown)
    private:
        static char
            path[];
};

```

The data member `path[]` is a *private static data member*. During the execution of the program, only *one* `Directory::path[]` exists, even though more than one object 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 never *initialized* by constructors. At most they are *modified*. The reason for this is that the static data members exist *before* any constructor of the class has been called. The static data members can be initialized during their definition, outside of all member functions, in the same way as global variables are initialized. 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 `path[]` from the above class `Directory` could thus be defined and initialized as follows in a file `data.cc`:

```

include "directory.H"

```

```
char
    Directory::path[200] = "/usr/local";
```

In the class interface the static member is actually only *declared*. At its implementation (definition) its type and class name are explicitly stated. Note also that the size specification can be left out of the interface, as is shown in the above array `path[]`. However, its size *is* (either explicitly or implicitly) needed at its definition.

Note that *any* source file could contain the definition of the static data members of a class. A separate `data.cc` source 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 `nobjects`. The variable `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
{
    public:
        Graphics();
        ~Graphics();           // other members not shown.
    private:
        void setgraphicsmode(); // switch to graphics mode
        void settextmode();     // switch to text-mode

        static int nobjects;    // counts # of objects
}
```

The purpose of the variable `nobjects` is to count the number of objects which exist at one given 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::nobjects = 0;           // the static data member

Graphics::Graphics()
{
    if (!nobjects++)
        setgraphicsmode();
}

Graphics::~Graphics()
{
    if (!--nobjects)
        settextmode();
}
```

Obviously, when the class `Graphics` would define more than one constructor, each constructor would need to increase the variable `nobjects` and would possibly have to initialize the graphics mode.

### 10.1.2 Public static data

Data members can be declared in the `public` section of a class, although this is not common practice (such a setup would violate the principle of data hiding). E.g., when the static data member `path[]` from section 10.1 would be declared in the `public` section of the class definition, all program code could access this variable:

```
int main()
{
    getcwd(Directory::path, 199);
}
```

Note that the variable `path` would still have to be defined. As before, the class interface would only *declare* the array `path[]`. This means that some source file would still need to contain the definition of the `path[]` array.

## 10.2 Static member functions

Besides static data members, C++ allows the definition of *static member functions*. Similar to the concept of static data, in which these variables are shared by all objects of the class, static member functions 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. Static member functions are themselves not associated with any object of their class. Consequently, they have not this pointer. In fact, a static member function is completely comparable to a global function, not associated with any class. This implies that the address of a static member function could be used in functions having parameters that are pointers to (global) functions. Since static member functions are comparable to ordinary global functions, static member functions that are declared in the `public` section of a class interface can be called without specifying an object of the class. The following example illustrates these characteristics of static member functions:

```
class Directory
{
    public:
        static void setpath(char const *newpath);
        static void preset(Directory &dir, char const *path);
    private:
        string
            currentPath;
        static char path[];
};

char Directory::path[200] = "/usr/local";           // see the text below
                                                    // 1

void Directory::setpath(char const *newpath)
{
    if (strlen(newpath) >= 200)
        throw "newpath too long";
}
```

```

        strcpy(path, newpath);                // 2
    }

    void Directory::preset(Directory &dir, char const *newpath)
    {
        dir.currentPath = 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 longer array is defined to be able to accomodate longer paths. Alternatively, a string or a pointer to dynamic memory could have been used.
- at 2 a (possibly longer, but not too long) new pathname is stored in the static data member `path[]`. Note that here only static members are used.
- at 3 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.
- 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 realized as in 4. But here `dir` is used to tell the compiler that we're talking about a function in the `Directory` class. So, static member functions *can* be called as normal member functions.
- at 6, the `currentPath` member of `dir` is altered. As in 4, the class and the scope resolution operator are used.
- at 7, the same is realized 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 private static member functions: these functions can only be called by member functions of their class.

Finally it is noted that static member functions cannot be defined as inline functions. Since a static function can be used as an ordinary function at the global level, it must have an address. The code of inline functions is normally directly inserted into the code of the program: not a true function, but an alias for a few statements. With inline functions we have no 'function address', Consequently, inline functions lack a basic requirement for static member functions: they have no address.

# Chapter 11

## Friends

In all examples we've discussed up to now, we've seen that `private` members are only accessible by the code of their class. This is *good*, as it enforces the principles of encapsulation and data hiding: By encapsulating the data in an object we can prevent that code external to classes becomes implementation dependent on the data in a class, and by hiding the data from external code we can control modifications of the data, helping us to maintain data integrity.

In this short chapter we will introduce the `friend` keyword as a means of allowing external functions to access the `private` members of a class. In this chapter the subject of friendship among classes is not discussed. Situations in which it is natural to use friendship among classes are discussed in chapters 16 and 18.

Friendship (i.e., using the `friend` keyword) is a complex and dangerous topic for various reasons:

- Friendship, when applied to program design, is an *escape mechanism* allowing us to circumvent the principles of encapsulation and data hiding. The use of friends should therefore be *minimized* to situations where they can be used naturally.
- If friends are used, realize that friend functions or classes become implementation dependent on the classes declaring them as friends. Once the internal organization of the data of a class declaring friends changes, all its friends must be recompiled (and possibly modified) as well.
- Therefore, as a rule of thumb: *don't* use friend functions or classes.

Nevertheless, there are situations where the `friend` keyword can be used quite safely and naturally. It is the purpose of this chapter to introduce the required syntax and to develop principles allowing us to recognize cases where the `friend` keyword can be used with very little danger.

Let's consider a situation where it would be nice for an existing class to have access to another class. Such a situation might occur when we would like to give a class developed *earlier* in history access to a class developed *later* in history.

Unfortunately, while developing the older class, it was not yet known that the newer class would be developed. Consequently, no provisions were offered in the older class to access the information in the newer class.

Consider the following situation. The insertion operator may be used to insert information into a stream. This operator can be given data of several types: `int`, `double`, `char *`, etc.. Earlier (chapter 7), we introduced the class `Person`. The class `Person` has members to retrieve the data stored in the `Person` object, like `char const *Person::getName()`. These members could be used to 'insert' a `Person` object into a stream, as shown in section 9.2.

With the `Person` class the implementation of the insertion and extraction operators is fairly optimal. The insertion operator uses *accessor* members which can be implemented as inline functions, effectively making the private data members directly available for inspection. The extraction operator requires the use of *modifier* members that could hardly be implemented differently: the old memory will always have to be deleted, and the new value will always have to be copied to newly allocated memory.

But let's once more take a look at the class `PersonData`, introduced in section 9.4. It seems likely that this class has at least the following (private) data members:

```
class PersonData
{
    private:
        Person
            *person;
        unsigned
            n;
};
```

When constructing an overloaded insertion operator for a `PersonData` object, e.g., inserting the information of all its persons, one on each line, into a stream, the overloaded insertion operator is not so well implemented when the individual persons have to be accessed using the index operator, which will not generally be an inline function.

In cases like these, where the accessor and modifier members tend to become rather complex member functions, direct access to the private data members might improve efficiency. So, in the context of insertion and extraction, we are in that case looking for overloaded member functions implementing the insertion and extraction operations that have access to the private data members of the objects to be inserted or extracted. In order to implement such functions *non-member* functions must be given access to the private data members of a class. This is realized by using the `friend` keyword.

## 11.1 Friend-functions

Concentrating on the `PersonData` class, the starting point for the insertion operator is the following:

```
ostream &operator<<(ostream &str, PersonData const &pd)
{
    for (unsigned idx = 0; idx < pd.nPersons(); idx++)
        str << pd[idx] << endl;
}
```

This implementation will perform its task as expected: using the (overloaded) insertion operator of the class `Person`, every `Person` stored in the `PersonData` object will be written to a separate line.

However, the repeated call to the index operator in the above implementation might reduce the efficiency of the implementation. Direct use of the array pointed to by `Person *person` might improve the efficiency of the above function.

At this point we should ask ourselves if we consider the above `operator<<()` primarily an extension of the globally available `operator<<()` function, or in fact a member function of the class `PersonData`. Stated otherwise: assume we would be able to make `operator<<()` into a true member function

of the class `PersonData`, would we object? Probably not, as the function's task is very closely tied to the class `PersonData`. In that case, the function can be made a *friend* of the class `PersonData`, thereby allowing the function access to the private data members of the class `PersonData`.

Friend functions must be declared as friends in the class interface. These *friend declarations* are in neither private nor public functions, and the friend declaration may therefore be placed anywhere in the class interface. Convention dictates that friend declarations are listed directly at the top of the class interface. So, for the class `PersonData` we get:

```
class PersonData
{
    friend ostream &operator<<(ostream &stream, PersonData &pd);
    friend istream &operator>>(istream &stream, PersonData &pd);

    public:
        // rest of the interface
};
```

The implementation of the insertion operator can now be altered in such a way that the insertion operator directly accesses the private data members of the provided `PersonData` object:

```
ostream &operator<<(ostream &str, PersonData const &pd)
{
    for (unsigned idx = 0; idx < pd.n; idx++)
        str << pd.person[idx] << endl;
}
```

Once again, whether friend functions are considered acceptable or not remain a matter of taste: if the function is in fact considered a member function, but due to the C++ grammar the function cannot be defined as a member function, then there seems to be little reason not to use the friend keyword. In other cases, we think the friend keyword should be avoided, thereby respecting the principles of *encapsulation* and *data hiding*.

Explicitly note that if we want to be able to insert `PersonData` objects into `ostream` objects without using the friend keyword, the insertion operator cannot be placed inside the `PersonData` class. In this case `operator<<()` is a normal overloaded variant of the insertion operator, which must therefore be declared and defined outside of the `PersonData` class. This situation applies, e.g., to the example at the beginning of this section.

## 11.2 Inline friends

In the previous section we stated that friends can be considered member functions of a class, albeit that the characteristics of the function prevents us from actually defining the function as a member function. In this section we will follow this line of thought a little further.

If we actually consider the friend function a member function, we should be able to design a true member function that performs the same tasks as our friend function. For example, we could construct a function that inserts a `PersonData` object into an `ostream`:

```
ostream &PersonData::insertor(ostream &str) const
{
```



```

        for (unsigned idx = 0; idx < n; idx++)
            str << person[idx] << endl;
        return str;
    }

```

This member function can be called with a `PersonData` object to insert that object into the ostream `str`:

```

PersonData
    pd;

cout << "The Person-information in the PersonData object is:\n";
pd.insertor(str);
cout << "=====\n";

```

Realizing that `insertor()` does the same thing as the overloaded insertor operator, that we defined as a friend, we could simply call the `insertor()` member in the code of the friend `operator<<()` function. Now this latter function has *only one statement*: it merely calls `insertor()`. Consequently:

- The `insertor()` function may be hidden in the class by making it private, as there is not need for it to be called elsewhere
- The `operator<<()` may be constructed as *inline* function, as it contains but one statement.

Thus, the relevant section of the class interface of `PersonData` becomes:

```

class PersonData
{
    friend ostream &operator<<(ostream &str, PersonData const &pd)
    {
        return pd.insertor(str);
    }
private:
    ostream &insertor(ostream &str) const;
};

```

The above example represents the final step in the development of friend functions. It allows us to formulate the following principle:

Although friend functions have access to private members of a class, this characteristic should not be used indiscriminately, as it results in a severe breach of the principle of encapsulation, thereby making non-class functions dependent on the implementation of the data in a class.

Instead, if the task a friend function performs, can be implemented in a true member function, it can be argued that a friend is merely a syntactical synonym or alias for this member function.

The interpretation of a friend function as a synonym for a member function is made concrete by constructing the friend function as an *inline* function.

As a principle we therefore state that friend functions should be avoided, unless they can be constructed as inline functions, having only one statement, in which an appropriate private member function is called.

Using this principle, we ascertain that all code that has access to the private data of a class remains confined to the class itself. This even holds true for `friend` functions, as they are defined as simple inline functions.

## 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 by the time the containers were constructed. That's why they are spoken of as *abstract containers*.

Abstract containers rely heavily on *templates*, which are covered near the end of the C++ Annotations, in chapter 18. However, in order to use the abstract containers, only a minimal grasp of the template concept is needed. 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 operate. As the types of the data on which the templates operate were not known by the time the template was constructed, the datatypes are either inferred from the context in which a template function is used, or they are mentioned explicitly by the time a template class is used (the term that's used here is *instantiated*). In situations where the types are explicitly mentioned, the *angular bracket notation* is used to indicate which data types are required. For example, below (in section 12.1) we'll encounter the *pair* container, which requires the explicit mentioning of two data types. E.g., to define a *pair* variable containing both an *int* and a *string*, the notation

```
pair<int, string>
myPair;
```

is used. Here, *myPair* is defined as a *pair* variable, containing both an *int* and a *string*.

The angular bracket notation is used intensively in the following discussion of abstract containers. Actually, understanding this part of templates is the only real requirement for using the abstract containers. Now that we've introduced this notation, we can postpone the more thorough discussion of templates to chapter 18, and get on with their use in the form of the abstract container classes.

Most of the abstract containers are *sequential* containers: they represent a series of data which can be stored and retrieved in some sequential way. Examples are the *vector*, implementing an extendable array, the *list*, implementing a datastructure in which insertions and deletions can be easily realized, a *queue*, also called a *FIFO* (first in, first out) structure, in which the first element that is entered will be the first element that will be retrieved, and the *stack*, which is a *first in, last*

*out* (FILO or LIFO) structure.

Apart from the 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 in which keys and corresponding values are stored. 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*.

All abstract containers described in this chapter and the `string` datatype discussed in chapter 4 are part of the Standard Template Library. There also exists an abstract container for the implementation of a *hashtable*, but that container is not (yet) accepted by the ANSI/ISO standard. Nevertheless, the final section of this chapter will cover the hashtable to some extent. It may be expected that containers like `hash_map` and other, now still considered an extension, will become part of the ANSO/ISO standard at the next release: apparently by the time the standard was frozen these containers were not yet fully available. Now that they are available they cannot be official part of the C++ library, but they are in fact available, albeit as extensions.

All containers support the following operators:

- The overloaded assignment operator, so we can assign two containers of the same type to each other.
- 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 container is less than each corresponding element in the right-hand container. If the right-hand container has more elements than the left-hand container, they are ignored. Other ordering operators perform analogously.

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 (`<`)

Closely linked to the standard template library are the *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 17. Generic algorithms usually rely on the availability of *iterators*, which represent begin and end-points for processing data stored within containers. The abstract containers normally have constructors and members expecting iterators for their arguments, and they often have members returning iterators (comparable to the `string::begin()` and `string::end()` members). In the remainder of this chapter the use of iterators is not really covered. Refer to chapter 17 for a discussion of iterators.

The url <http://www.sgi.com/Technology/STL> is worth visiting by those readers who are looking for more information about the abstract containers and the standard template library than can be provided in the C++ annotations.

Containers often collect data during their lifetime. 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, the data to which these pointers point will not be destroyed, and a memory leak will result. A consequence of this scheme is that the data stored in a container should be considered the ‘property’ of the container: the container should be able to destroy its data elements when the destructor of the container is called. Consequently, in normal circumstances the container should contain no pointer data. Also, a container should not be required to contain `const` data, as for `const` data the, e.g., `operator=()` doesn't work.

Below, at the descriptions of the containers, the following notational convention is used:

- A container without angular brackets represents any container of that type. Mentally add the required type in angular bracket notation. E.g., `pair` may represent `pair<string, int>`.
- The notation `Type` represents the generic type. `Type` could be `int`, `string`, etc.
- Identifiers `object` and `container` represent objects of the container type under discussion.
- The identifier `value` represents a value of the type that is stored in the container.
- Simple, one-letter identifiers, like `n` represent unsigned values.
- Longer identifiers represent iterators. Examples are `pos`, `from`, `beyond`

Some containers, e.g., the `map` container, contain combinations of values, usually called ‘keys’ and ‘values’. For such containers the following notational convention is used in addition:

- The identifier `key` indicates a value of the used key-type
- The identifier `keyvalue` indicates a value of the ‘value\_type’ used with the particular container.

## 12.1 The ‘pair’ container

The `pair` container is a rather basic container. It can be used to store two elements, called `first` and `second`, and that's about it. To define a `pair` container, source files should

```
#include <utility>
```

The data types of a `pair` are defined when the `pair` variable is defined, using the standard template (see chapter `Templates`) angular bracket notation:

```
pair<string, string>
  piper("PA28", "PH-ANI"),
  cessna("C172", "PH-ANG");
```

here, the variables `piper` and `cessna` are defined as `pair` variables containing two `strings`. Both `strings` can be retrieved using the `first` and `second` fields of the `pair` type:

```
cout << piper.first << endl <<      // shows 'PA28'
      cessna.second << endl;         // 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 side 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 containername pair. It also requires the specification of the data types which are stored within the pair. For this the (template) angular 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 a lot of pair<type1, type2> clauses are used in a source, the typing effort may be reduced and legibility 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.

## 12.2 Sequential Containers

### 12.2.1 The 'vector' container

The vector class implements an expandable array. To use the vector, source files should

```
#include <vector>
```

The following constructors, operators, and member functions are available:

- Constructors:
  - A vector may be constructed empty:

```
vector<string>  
object;
```

Note the specification of the data type to be stored in the vector: the data type is given between angular brackets, just after the 'vector' container name. This is common practice with containers.

- A vector may be initialized to having a certain number of elements. One of the nicer characteristics of vectors (and other containers) is that it initializes its 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 the int vector we know its initial values are zero. For example:

```
vector<string>
    object(5, string("Hello")), // initialize to 5 Hello's,
    container(10);              // and to 10 empty strings
```

- A vector may be initialized using a two iterators. To initialize a vector with elements 5 until 10 (including the last one) of a 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 last value mentioned. The standard notation for this is [begin, end).

- A vector may be initialized using a copy constructor:

```
extern vector<string>
    container;
vector<string>
    object(container);
```

- Apart from the standard operators for containers, the vector supports the index operator, which may 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 it *does* respect its array bounds. In this case the vector should be resized first, or `ivect.push_back(18)` should be used (see below).

- The vector class has the following member functions:

- `Type &vector::back()`:  
this member returns a reference to the last element in the vector. It is the responsibility of the programmer not to use the member if the vector is empty.
- `vector::iterator vector::begin()`:  
this member returns an iterator pointing to the first element in the vector.
- `vector::clear()`:  
this member erases all elements in the vector.
- `bool vector::empty()`:  
this member returns true if the vector contains no elements.
- `vector::iterator vector::end()`:  
this member returns an iterator pointing beyond the last element in the vector.
- `vector::iterator vector::erase()`:  
this member can be used to erase a specific range of elements in the vector:

- \* `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 &vector::front()`:  
this member returns a reference to the first element in the vector. It is the responsibility of the programmer not to use the member if the vector is empty.
- `... vector::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`.
- `void vector::pop_back()`:  
this member removes the last element from the vector. With an empty vector nothing happens.
- `void vector::push_back(value)`:  
this member adds `value` to the end of the vector.
- `void vector::resize()`:  
this member 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 `tvalue` is not provided, the extra elements are initialized to the default value of the used data type, otherwise the explicitly provided value `value` is used to initialize extra elements.
- `vector::reverse_iterator vector::rbegin()`:  
this member returns an iterator pointing to the last element in the vector.
- `vector::reverse_iterator vector::rend()`:  
this member returns an iterator pointing before the first element in the vector.
- `unsigned vector::size()`  
this member returns the number of elements in the vector.
- `void vector::swap()`  
this member can be used to swap two vectors. E.g.,
 

```
#include <iostream>
#include <vector>

int main()
{
    vector<int>
        v1(7),
        v2(10);

    v1.swap(v2);
    cout << v1.size() << " " << v2.size() << endl;
}
```



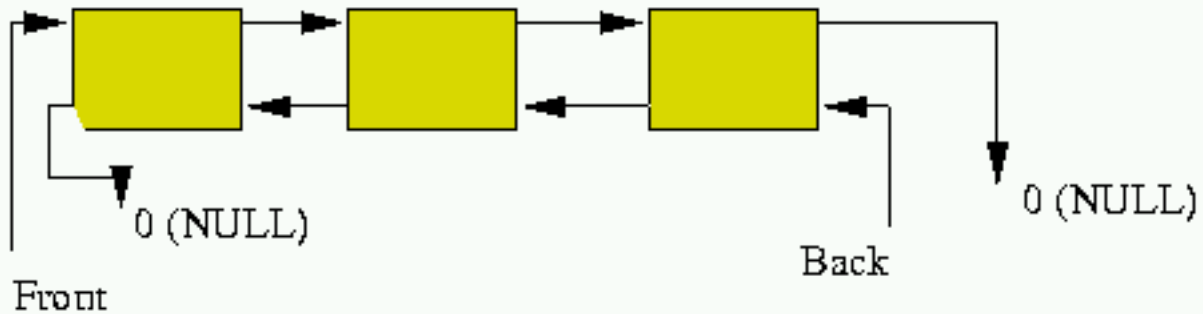


Figure 12.1: A list data-structure

```
/*
    Produced output:
    10 7
*/
```

### 12.2.2 The 'list' container

The `list` container implements a list data structure. To use the `list`, source files should

```
#include <list>
```

The organization of a `list` is shown in figure 12.1.

In figure 12.1 it is shown that a list consists of separate list-elements, connected to each other by pointers. The list can be traversed in two ways: 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.

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 a choice between the two data structures must be made.

- When the majority of accesses is random, then the `vector` is the preferred data structure. E.g., in a program that counts the frequencies of characters in a textfile, a `vector<int> frequencies(256)` is the datastructure doing the trick, 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, the list is generally preferred. Actually, in my experience, lists aren't that useful at all, and often an implementation will be faster when a `vector`, maybe containing holes, is used.

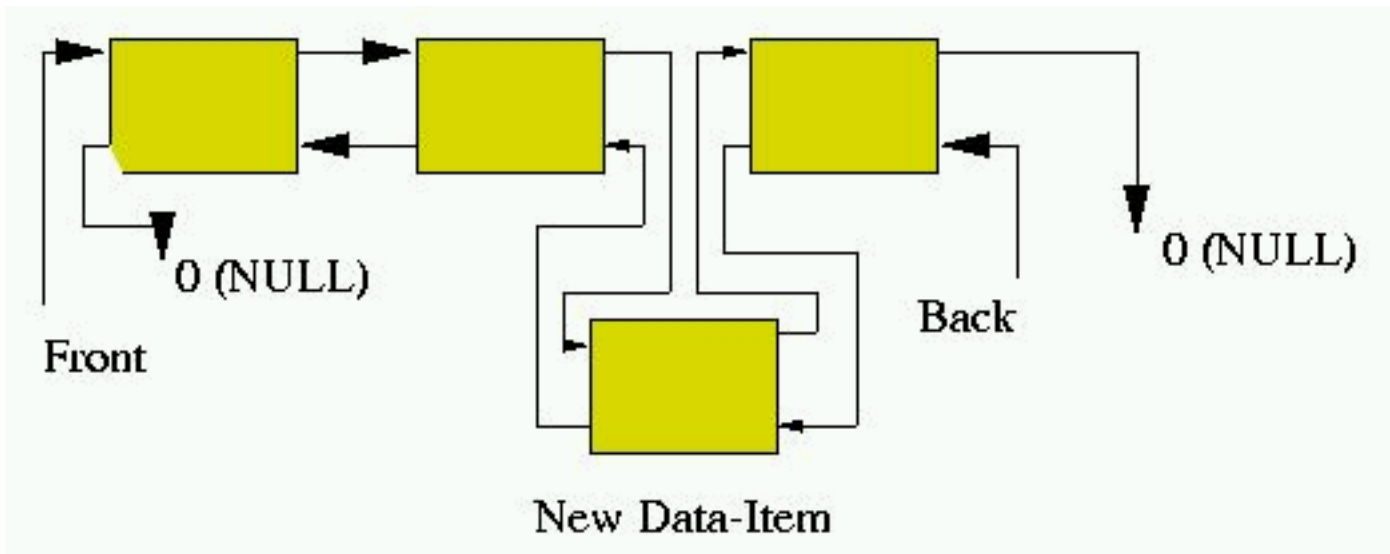


Figure 12.2: Adding a new element to a list

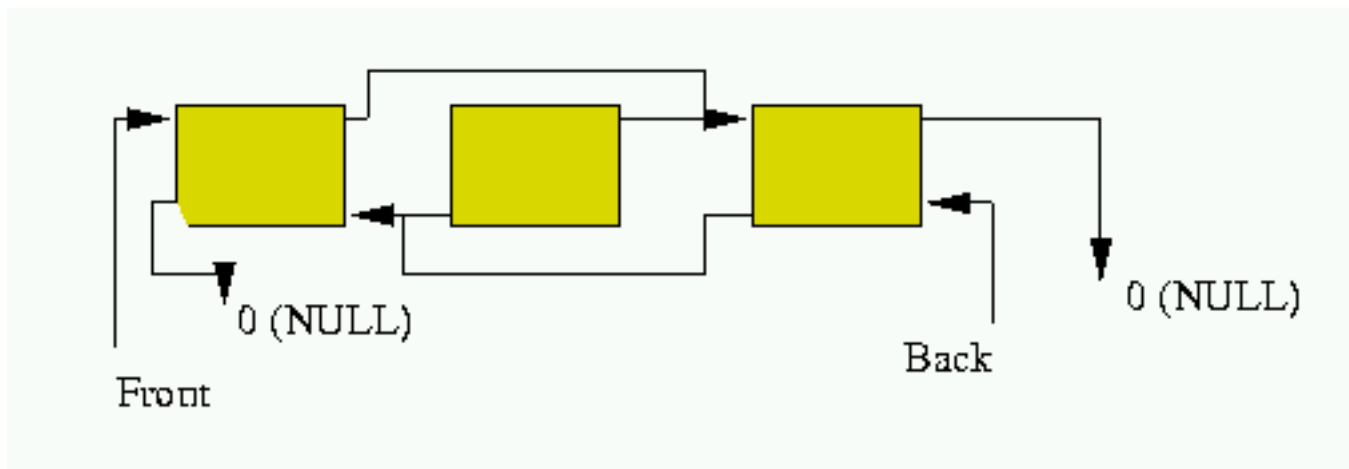


Figure 12.3: Removing an element from a list

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 does involve a lot 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 also is a simple matter. 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.

Summarizing 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

working best.

But, no matter what the thoughts on the subject are, the `list` container is available, so let's see what we can do with it. The following constructors, operators, and member functions are available:

- Constructors:

- 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 having 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:

```
list<string>
    object(5, string("Hello")), // initialize to 5 Hello's
    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:

```
extern vector<string>
    container;
list<string>
    object(&container[5], &container[11]);
```

- A list may be initialized using a copy constructor:

```
extern list<string>
    container;
list<string>
    object(container);
```

- There are no special operators available for the `list`, apart from the standard operators for containers.

- The following member functions are available for lists:

- Type `&list::back()`:

this member returns a reference to the last element in the list. It is the responsibility of the programmer not to use the member if the list is empty.

- `list::iterator list::begin()`:

this member returns an iterator pointing to the first element in the list.

- `list::clear()`:

this member erases all elements in the list.

- `bool list::empty()`:

this member returns `true` if the list contains no elements.

- `list::iterator list::end()`:

this member returns an iterator pointing beyond the last element in the list.

- `list::iterator list::erase()`:

this member can be used to erase 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 &list::front()`:  
this member returns a reference to the first element in the list. It is the responsibility of the programmer not to use the member if the list is empty.
- ... `list::insert()`:  
this member can be used to insert 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`.
- `void list<Type>::merge(list<Type> other)`:  
this member function assumes that the current and other lists are sorted (see below, the member `sort()`), and will, based on that assumption, insert 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'.

```
#include <iostream>
#include <string>
#include <list>

void showlist(list<string> &target)
{
    for
    (
        list<string>::iterator from = target.begin();
        from != target.end();
        ++from
    )
        cout << *from << " ";

    cout << endl;
}

int main()
{
    list<string>
        first,
        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 list::pop_back()`:  
this member removes the last element from the list. With an empty list nothing happens.
- `void list::pop_front()`:  
this member removes the first element from the list. With an empty list nothing happens.
- `void list::push_back(value)`:  
this member adds `value` to the end of the list.
- `void list::push_front(value)`:  
this member adds `value` before the first element of the list.
- `void list::resize()`:  
this member can be used to alter 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 `tvalue` is not provided, the extra elements are initialized to the default value of the used data type, otherwise the explicitly provided value `value` is used to initialize extra elements.
- `list::reverse_iterator list::rbegin()`:  
this member returns an iterator pointing to the last element in the list.
- `void list::remove(value)`:  
this member 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 <string>
#include <list>

int main()
{
    list<string>
        object;

    object.push_back(string("Hello"));
    object.push_back(string("World"));
    object.push_back(string("Hello"));
    object.push_back(string("World"));

    object.remove(string("Hello"));

    while (object.size())

```

```

        {
            cout << object.front() << endl;
            object.pop_front();
        }
    }
    /*
        Generated output:
        World
        World
    */
- list::reverse_iterator list::rend():
    this member returns an iterator pointing before the first element in the list.
- unsigned list::size():
    this member returns the number of elements in the list.
- void list::reverse():
    this member reverses the order of the elements in the list. The element back()
    will become front() and vice versa.
- void list::sort():
    this member will sort 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 there-
    fore be available.
- void list::splice(pos, object):
    this member function transfers the contents of value to the current list. Following
    splice(), value is empty. For example:
#include <iostream>
#include <string>
#include <list>

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," << endl;

    while (object.size())
    {
        cout << object.front() << endl;
        object.pop_front();
    }
}

```

Alternatively, value may be followed by a iterator of value, indicating the first element of value that should be spliced, or by two iterators begin and end defining the iterator-range [begin, end) on value that should be spliced into target.

– void list::swap():

this member can be used to swap two lists.

– void list::unique():

operating on a sorted list, this member function will remove 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 that removes all doubly occurring words from the list:

```
#include <iostream>
#include <string>
#include <list>

// see the merge() example
void showlist(list<string> &target);

int main()
{
    string
        array[] =
        {
            "charley",
            "alpha",
            "bravo",
            "alpha"
        };

    list<string>
        target
        (
            array, array + sizeof(array)
            / sizeof(string)
        );

    cout << "Initially we have: " << endl;

        // shows: charley alpha bravo alpha
    showlist(target);

    target.sort();

    cout << "After sort() we have: " << endl;

        // shows: alpha alpha bravo charley
    showlist(target);

    target.unique();

    cout << "After unique() we have: " << endl;

        // shows: alpha bravo charley
    showlist(target);
}
```

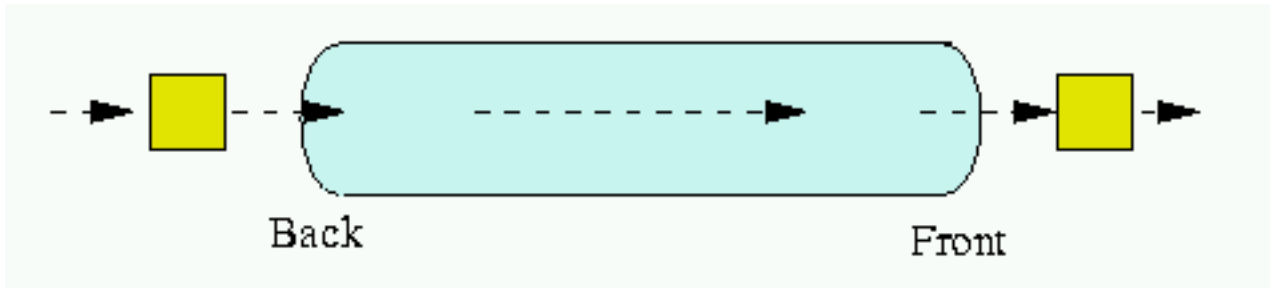


Figure 12.4: A queue data-structure

### 12.2.3 The 'queue' container

The `queue` class implements a queue data structure. To use the `queue`, source files should

```
#include <queue>
```

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:

- A queue may be constructed empty:

```
queue<string>
object;
```

As with the `vector`, it is an error to refer to an element of an empty queue.

- A queue may be initialized using a copy constructor:

```
extern queue<string>
container;
queue<string>
object(container);
```

- No other operators than the basic operators for containers are available.

- The following member functions are available for queues:

- Type `&queue::back()`:

this member returns a reference to the last element in the queue. It is the responsibility of the programmer not to use the member if the queue is empty.

- `bool queue::empty()`:

this member returns `true` if the queue contains no elements.

- Type `&queue::front()`:

this member returns a reference to the first element in the queue. It is the responsibility of the programmer not to use the member if the queue is empty.



- `void queue::push(value):`  
this member adds `value` to the back of the queue.
- `void queue::pop():`  
this member removes the element at the front of the queue. Note that the element is *not* returned by this member.
- `unsigned queue::size():`  
this member 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, and a queue can only be emptied by repeatedly removing its front element (or, of course, by having its destructor called).

### 12.2.4 The ‘priority\_queue’ container

The `priority_queue` class implements a priority queue data structure. To use the priority queue, source files should

```
#include <queue>
```

A priority queue is identical to a queue, but allows the entry of data elements according to priority rules. An example of a situation where the priority queue is encountered in real-life is found at the check-in terminals at airports. 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`:

```
#include <iostream>
#include <string>
#include <queue>

int main()
{
    priority_queue<string>
        q;
    string
        word;

    while (cin >> word)
        q.push(word);

    while (q.size())
    {
        cout << q.top() << endl;
        q.pop();
    }
}
```

Unfortunately, the words are listed in reversed order: because of the underlying <-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, in which the `operator<()` has been defined according to our wish, i.e., making sure that the words appearing early in the ASCII-sequence will appear first in the queue. Here is the modified program:

```
#include <iostream>
#include <string>
#include <queue>

class Text
{
public:
    Text(string const &str)
    :
        s(str)
    {}
    operator string const &() const
    {
        return s;
    }
    bool operator<(Text const &right) const
    {
        return s > right.s;
    }
private:
    string
        s;
};

int main()
{
    priority_queue<Text>
        q;
    string
        word;

    while (cin >> word)
        q.push(word);

    while (q.size())
    {
        word = q.top();
        cout << word << endl;
        q.pop();
    }
}
```

In the above program the wrapper class defines the `operator<()` just the other way around than the `string` class itself, resulting in the preferred ordering. Other possibilities 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:

- 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`.

- A `priority_queue` may be initialized using a copy constructor:

```
extern priority_queue<string>
    container;
priority_queue<string>
    object(container);
```

- The `priority_queue` only supports the basic operators of containers.

- The following member functions are available for `priority_queues`:

- `bool priority_queue::empty()`:  
this member returns `true` if the `priority_queue` contains no elements.
- `void priority_queue::push(value)`:  
this member inserts `value` at the appropriate position in the `priority_queue`.
- `void priority_queue::pop()`:  
this member removes the element at the top of the `priority_queue`. Note that the element is *not* returned by this member.
- `unsigned priority_queue::size()`:  
this member returns the number of elements in the `priority_queue`.
- `Type &priority_queue::top()`:  
this member returns a reference to the first element of the `priority_queue`. It is the responsibility of the programmer not to use the member if the `priority_queue` is empty.

Note that the `priority_queue` does not support iterators or a subscript operator. The only elements that can be accessed are its top element, and a `priority_queue` can only be emptied by repeatedly removing its top element (or, of course, by having its destructor called).

### 12.2.5 The ‘deque’ container

The `deque` (pronounce: ‘deck’) class implements a doubly ended queue data structure (`deque`). To use the `deque` class, source files should

```
#include <deque>
```

A *deque* is comparable to a *queue*, but it allows reading and writing at both ends. Actually, the `deque` data type supports a lot more functionality than the `queue`, as will be clear from the following overview of member functions that are available for the `deque`. 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:

- A deque may be constructed empty:

```
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 having 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:

```
deque<string>
    object(5, string("Hello")), // initialize to 5 Hello's
    container(10);              // and to 10 empty strings
```

- A deque may be initialized using a 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:

```
extern vector<string>
    container;
deque<string>
    object(&container[5], &container[11]);
```

- A deque may be initialized using a copy constructor:

```
extern deque<string>
    container;
deque<string>
    object(container);
```

- Apart from 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 elements which are indexed must exist.

- The following member functions are available for deques:

- Type &deque::back():

this member returns a reference to the last element in the deque. It is the responsibility of the programmer not to use the member if the deque is empty.

- deque::iterator deque::begin():

this member returns an iterator pointing to the first element in the deque.

- deque::clear():

this member erases all elements in the deque.

- bool deque::empty():

this member returns true if the deque contains no elements.

- deque::iterator deque::end():

this member returns an iterator pointing beyond the last element in the deque.

- deque::iterator deque::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 &deque::front()`:  
this member returns a reference to the first element in the deque. It is the responsibility of the programmer not to use the member if the deque is empty.
- `... deque::insert()`:  
this member can be used to insert 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` at position `pos`.
- `void deque::pop_back()`:  
this member removes the last element from the deque. With an empty deque nothing happens.
- `void deque::pop_front()`:  
this member removes the first element from the deque. With an empty deque nothing happens.
- `void deque::push_back(value)`:  
this member adds `value` to the end of the deque.
- `void deque::push_front(value)`:  
this member adds `value` before the first element of the deque.
- `void deque::resize()`:  
this member can be used to alter 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 `tvalue` is not provided, the extra elements are initialized to the default value of the used data type, otherwise the explicitly provided value `value` is used to initialize extra elements.
- `deque::reverse_iterator deque::rbegin()`:  
this member returns an iterator pointing to the last element in the deque.
- `deque::reverse_iterator deque::rend()`:  
this member returns an iterator pointing before the first element in the deque.
- `unsigned deque::size()`:  
this member returns the number of elements in the deque.
- `void deque::swap()`:  
this member can be used to swap two deques.

## 12.2.6 The 'map' container

The `map` class implements a (sorted) associative array. To use the `map`, source files should

```
#include <map>
```

A `map` is filled with *key/value* pairs, which may be of any container-acceptable type. Since types are associated with both the key and the value, we must specify two types in the angular bracket notation that is used for maps. The first type represents the type of the key, the second type represents the type of the value. For example, a `map` in which the key is a `string` and the value is a `double` can be defined as follows:

```
map<string, double>
    object;
```

The *key* is used for locating the information belonging to the key. The associated 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 the value.

The two fundamental operations on maps are the storage of *Key/Value* combinations, and the retrieval of values, given their keys. 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 be implicitly or explicitly 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:

```
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.,

```
map<string, int>::value_type("Hello", 1);
```

Instead of using the line `map<string, int>::value_type(...)` over and over again, a `typedef` is often used to reduce typing and to improve legibility:

```
typedef map<string, int>::value_type StringIntValue
```

Using this `typedef`, values for the `map<string, int>` may be constructed as follows:

```
StringIntValue("Hello", 1);
```

The following constructors, operators, and member functions are available for the `map` container:

- Constructors:
  - 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 in another container:

```
map<string, pair<string, string> >
    object;
```

Note the blank space between the two closing angular brackets >: this is obligatory, as the immediate concatenation of the two angular closing brackets would be interpreted by the compiler as a right shift operator (`operator>>()`), which is not what we want here.

- A map may be initialized using a two iterators. The iterators may point to `value_type` values for the map to be constructed, or they may point to plain pair objects (see section 12.1), whose first elements represent the keys, and whose second elements represent the values to be used. For 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 used instead of `pair<string, int>` as well.

Note again that `&pa[3]`, being interpreted as an iterator, points to the first element that must *not* be included in the map. The particular array element does not have to exist.

Also note that, maybe contrary to intuition, the map constructor will only enter *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 have been silently ignored.

Finally, it is worth noting that the map receives its own copies of the data to which the iterators point. This is illustrated by the following example:

```
#include <iostream>
#include <string>
#include <map>

class MyClass
{
public:
    MyClass()
    {
        cout << "MyClass constructor\n";
    }
    MyClass(const MyClass &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
*/

```

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 needed anymore, and gets 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 element. Having constructed the map element, the temporary pair objects is destroyed. Eventually, when the program terminates, the pair element stored in the map is destroyed too.

- A map may be initialized using a copy constructor:

```

extern map<string, int>
    container;
map<string, int>
    object(container);

```

- Apart from the standard operators for containers, the map supports the index operator, which may be used to retrieve or reassign individual elements of the map. Here, the argument of the index operator is 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 right-hand side of the assignment operator must have the type of the value part of the map. E.g., to add or change the value of element "two" in a map, the following statement can be used:

```

mapsm["two"] = MyClass();

```



- The map class has the following member functions:

- `map::iterator map::begin()`:  
this member returns an iterator pointing to the first element of the map.
- `map::clear()`:  
this member erases all elements from the map.
- `unsigned map::count(key)`:  
this member returns 1 if the provided key is available in the map, otherwise 0 is returned.
- `bool map::empty()`:  
this member returns true if the map contains no elements.
- `map::iterator map::end()`:  
this member returns an iterator pointing beyond the last element of the map.
- `pair<map::iterator, map::iterator> map::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 the use of these member functions is given at the discussion of the member function `upper_bound()`.
- ... `map::erase()`:  
this member can be used to erase 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 returned 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 pos.
  - \* `void erase(first, beyond)` erases all elements indicated by the iterator range [first, beyond).
- `map::iterator map::find(key)`:  
this member returns an iterator to the element having the given key. If the element isn't available, `end()` is returned. The following example illustrates the use of the `find()` member function:

```
#include <iostream>
#include <string>
#include <map>

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
    */

```

– ... map::insert():

this member can be used to insert elements starting at a certain position. Elements having the same keys as elements to be inserted are left untouched. The return value depends on the version of insert() that is called:

\* pair<map::iterator, bool> insert(keyvalue) is used to insert a new map::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 in the map having the key that was specified in keyvalue. The use of this variant of insert() is illustrated in the following example:

```

#include <iostream>
#include <string>
#include <map>

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, 4} and 'true' (1) is returned
    pair<map<string, int>::iterator, bool>
        ret = object.insert
        (
            map<string, int>::value_type
            ("four", 40)
        );

    cout << ret.first->first << " " <<
        ret.first->second << " " <<
        ret.second << " " << object["four"] << endl;

    // {four, 4} and 'false' (0) is returned
    ret = object.insert
    (
        map<string, int>::value_type
        ("four", 0)
    );
}

```

```

    );

    cout << ret.first->first << " " <<
         ret.first->second << " " <<
         ret.second << " " << object["four"] << endl;
}
/*
    Generated output:
    four 40 1 40
    four 40 0 40
*/

```

Note the somewhat peculiar constructions like

```
cout << ret.first->first << " " << ret.first->second << ...
```

Realize that 'ret' is the pair variable 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 is another way to insert a map::value\_type 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).

- map::iterator map::lower\_bound(key):

this member returns an iterator pointing to the element with the given key. If no such value exists, map::end() is returned.

- map::reverse\_iterator map::rbegin():

this member returns an iterator pointing to the last element of the map.

- map::reverse\_iterator map::rend():

this member returns an iterator pointing before the first element of the map.

- unsigned map::size():

this member returns the number of elements in the map.

- void map::swap():

this member can be used to swap two maps.

- map::iterator map::upper\_bound(key):

this member returns the iterator map::end(). The following example illustrates the member functions equal\_range(), lower\_bound() and upper\_bound():

```

#include <iostream>
#include <string>
#include <map>

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]);

    if (object.lower_bound("two") == object.end())
        cout << "lower-bound 'two' not available" << endl;

    cout << "lower-bound two: " <<
        object.lower_bound("two")->first <<
        " is available\n";

    if (object.upper_bound("two") == object.end())
        cout << "upper-bound 'two' not available" << endl;

    if (object.upper_bound("two") == object.end())
        cout << "upper-bound 'two' not available" << endl;

    pair
    <
        map<string, int>::iterator,
        map<string, int>::iterator
    >
        p = object.equal_range("two");

    cout << "equal range: 'first' points to " <<
        p.first->first << ", 'second' is " <<
        (
            p.second == object.end() ?
                "not available"
            :
                p.second->first
        ) <<
        endl;
}
/*
    Generated output:
lower-bound 'two' not available
lower-bound two: two is available
upper-bound 'two' not available
upper-bound 'two' not available
equal range: 'first' points to two, 'second' is not available
*/

```

As mentioned at the beginning of this section, the map represents a sorted associative array. In a map the keys are sorted. If an application must visit all elements in a map (or just the keys or the values) the `begin()` and `end()` iterators must be used. The following example shows how to make a simple table from all keys and values in a map:

```

#include <iostream>
#include <iomanip>
#include <string>
#include <map>

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 << endl;
}
/*
    Generated output:
    one    10
    three  30
    two    20
*/

```

### 12.2.7 The ‘multimap’ container

Like the map, the multimap class implements a (sorted) associative array. To use the multimap, source files must

```
#include <map>
```

The main difference between the map and the multimap is that the multimap supports multiple entries of the same key, whereas the map contains unique keys. Note that the multimap also accepts multiple entries of values having the same keys *and* the same values.

The map and the multimap have the same set of member functions, with the exception of the index operator (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 12.2.6 for an overview of member functions that are available with the multimap. Some member functions, however, act a bit different with the multimap than with the map. These differences are discussed below.

The following member functions act differently with multimap containers than with map containers:

- unsigned map::count(key):  
this member returns the number of entries in the multimap that are associated with the given key.
- ... multimap::erase():  
this member can be used to erase elements from the map:

- `unsigned erase(key)` erases all elements having the given `key`. The number of erased elements is returned.
  - `void erase(pos)` erases the 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> multimap::equal_range(key):`  
 this member function returns a pair of iterators, being respectively the return values of `multimap::lower_bound()` and `multimap::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 multimap::find(key):`  
 this member returns an iterator pointing to the first value whose key is `key`. If the element isn't available, `multimap::end()` is returned. The iterator could be incremented to visit all elements having the same key until it is either `multimap::end()`, or the iterator's first member is not equal to `key` anymore.
  - `... multimap::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.
  - `multimap::iterator multimap::lower_bound(key):`  
 this member returns an iterator pointing to the first keyvalue element of which the key is equal to or exceeds the specified key. If no such element exists, the function returns `multimap::end()`.
  - `multimap::iterator multimap::upper_bound(key):`  
 this member returns an iterator pointing to the first keyvalue element having a key exceeding the specified key. If no such element exists, the function returns `multimap::end()`.

The following example illustrates `multimap::lower_bound()`, `multimap::upper_bound()` and `multimap::equal_range`:

```
#include <iostream>
#include <string>
#include <map>

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': " <<
        it->first << ", " << it->second << endl;

    it = object.upper_bound("bravu");

    cout << "Upper bound for 'bravu': " <<
        it->first << ", " << it->second << endl;

    pair<msiIterator, msiIterator>
        itPair = object.equal_range("bravo");

    for (it = itPair.first; it != itPair.second; ++it)
        cout << it->first << ", " << it->second << endl;

    cout << "Upper bound: " << it->first << ", " << it->second << endl;
}
/*
    Generated output:
    Lower bound for 'brava': bravo, 2
    Upper bound for 'bravu': charley, 3
    bravo, 2
    bravo, 6
    bravo, 4
    Upper bound: charley, 3
*/

```

Especially 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.

## 12.2.8 The 'set' container

The `set` class implements a sorted collection of values. To use the `set`, source files must `#include <set>`

```
#include <set>
```

A set is filled with values, which may be of any container-acceptable type. Each value can be stored only once in a set.

A specific key/value combination can be implicitly or explicitly inserted into a `set`. If explicit insertion is required, the value must be constructed first. For this, 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 legibility:

```
typedef set<string>::value_type StringSetValue
```

Using this `typedef`, values for the `set<string>` may be constructed as follows:

```
StringSetValue("Hello");
```

The following constructors, operators, and member functions are available for the `set` container:

- Constructors:

- A `set` may be constructed empty:

```
set<int>  
    object;
```

- A `set` may be initialized using a 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 will be entered, the other values will be silently ignored.

Like the `map`, the `set` receives its own copy of the data it contains.

- A `set` may be initialized using a copy constructor:

```
extern set<string>  
    container;  
set<string>  
    object(container);
```

- 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 set::begin()`:  
this member returns an iterator pointing to the first element of the set.
- `set::clear()`:  
this member erases all elements from the set.
- `unsigned set::count(key)`:  
this member returns 1 if the provided key is available in the set, otherwise 0 is returned.
- `bool set::empty()`:  
this member returns true if the set contains no elements.
- `set::iterator set::end()`:  
this member returns an iterator pointing beyond the last element of the set.
- `pair<set::iterator, set::iterator> set::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.
- ... `set::erase()`:  
this member can be used to erase 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 pos.
  - \* `void erase(first, beyond)` erases all elements indicated by the iterator range [first, beyond).
- `set::iterator set::find(value)`:  
this member returns an iterator to the element having the given value. If the element isn't available, `end()` is returned.
- ... `set::insert()`:  
this member can be used to insert 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)` is used to insert 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 is another way to insert a `set::value_type` 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.
- `set::iterator set::lower_bound(key)`:  
this member returns an iterator pointing to the element with the given key. If no such value exists, `end()` is returned.
- `set::reverse_iterator set::rbegin()`:  
this member returns an iterator pointing to the last element of the set.

- `set::reverse_iterator set::rend()`:  
this member returns an iterator pointing before the first element of the set.
- `unsigned set::size()`:  
this member returns the number of elements in the set.
- `void set::swap()`:  
this member can be used to swap two sets.
- `set::iterator set::upper_bound(key)`:  
this member returns an iterator pointing to the element with the given key. If no such value exists, `end()` is returned.

### 12.2.9 The ‘multiset’ container

Like the `set`, the `multiset` class implements a sorted collection of value. To use the `multiset`, source files must

```
#include <set>
```

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 member functions. Refer to section 12.2.8 for an overview of member functions that are available with the `multiset`. Some member functions, however, act a bit different with the `multiset` than with the `set`. These differences are discussed below.

The following member functions act differently with `multiset` containers than with `set` containers:

- `unsigned set::count(value)`:  
this member returns the number of entries in the `multiset` that are associated with the given value.
- ... `multiset::erase()`:  
this member can be used to erase elements from the set:
  - `unsigned 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 `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> multiset::equal_range(value)`:  
this member function returns a pair of iterators, being respectively the return values of `multiset::lower_bound()` and `multiset::upper_bound()`, introduced below. The function provides a simple means to determine all elements in the `multiset` that have the same values.

- `multiset::iterator multiset::find(value):`  
 this member returns an iterator pointing to the first element having the specified value. If the element isn't available, `multiset::end()` is returned. The iterator could be incremented to visit all elements having the given value until it is either `multiset::end()`, or the iterator doesn't point to 'value' anymore.
- ... `multiset::insert():`  
 this member function normally succeeds, and so a *multiset::iterator* is returned, instead of a `pair<multiset::iterator, bool>` as returned with the `set` container. The returned iterator points to the newly added element.
- `multiset::iterator multiset::lower_bound(value):`  
 this member returns an iterator pointing to the first valuevalue element of which the value is equal to or exceeds the specified value. If no such element exists, the function returns `multiset::end()`.
- `multiset::iterator multiset::upper_bound(value):`  
 this member returns an iterator pointing to the first valuevalue element having a value exceeding the specified value. If no such element exists, the function returns `multiset::end()`.

An example showing the use of various member functions of a multiset is given next:

```
#include <iostream>
#include <string>
#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 << endl;

        pair
        <
            multiset<string>::iterator,
            multiset<string>::iterator
        >
            itpair = object.equal_range("echo");

        for (; itpair.first != itpair.second; ++itpair.first)
            cout << *itpair.first << " ";

        cout << endl <<
            object.count("echo") << " occurrences of 'echo'" << endl;
    }
    /*
        Generated output:
        echo echo echo hotel mike romeo
        echo echo echo
        3 occurrences of 'echo'
    */

```

### 12.2.10 The 'stack' container

The `stack` class implements a stack data structure. To use the `stack`, source files must

```
#include <stack>
```

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: these variables lifetime live only as long as the functions live, 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 expressions are entered in the stack, and the operators pop their operands and push the results of their work.

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  $(3 + 4) * 2$  is evaluated. In the RPN this expression becomes  $3\ 4\ +\ 2\ *$ , 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  $*$  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.

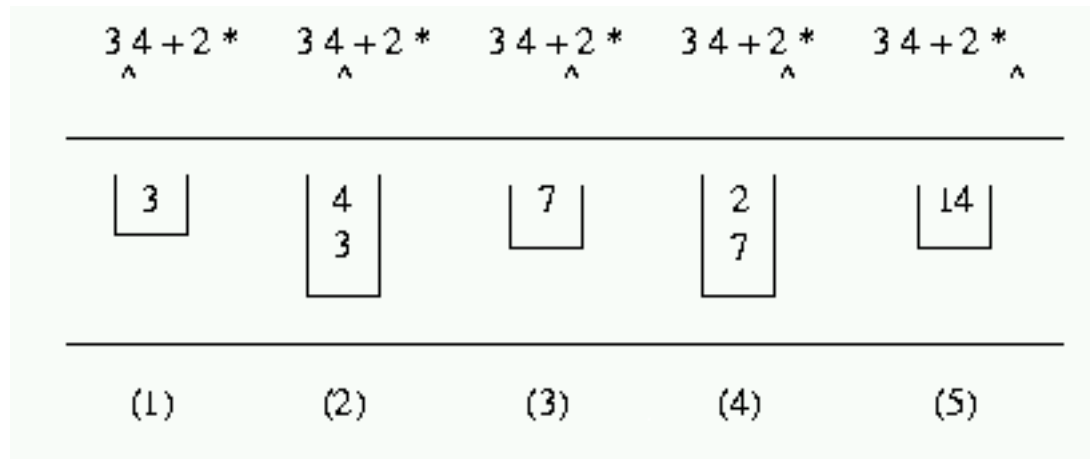


Figure 12.5: The contents of a stack while evaluating  $3\ 4 + 2 *$

From figure 12.5 we see that a stack has one point (the *top*) where items can be added to and removed from the stack. Furthermore, values can be pushed and popped from a stack.

Bearing this model of the stack in mind, let's see what we can formally do with it, using the `stack` container. For the `stack`, the following constructors, operators, and member functions are available:

- Constructors:
  - A stack may be constructed empty:
 

```
stack<string>
  object;
```
  - A stack may be initialized using a copy constructor:
 

```
extern stack<string>
  container;
stack<string>
  object(container);
```
- The `stack` only supports the basic operators of containers.
- The following member functions are available for stacks:
  - `bool stack::empty():`  
this member returns `true` if the stack contains no elements.
  - `void stack::push(value):`  
this member places `value` at the top of the stack, hiding the other elements from view.
  - `void stack::pop():`  
this member removes the element at the top of the stack. Note that the popped element is *not* returned by this member.
  - `unsigned stack::size():`  
this member returns the number of elements in the stack.
  - `Type &stack::top():`  
this member returns a reference to the first element of the stack. It is the responsibility of the programmer not to use the member if the stack is empty.

Note that the stack does not support iterators or a subscript operator. The only elements that can be accessed is its top element, and a stack can only be emptied by repeatedly popping its top-element.

### 12.2.11 The ‘hash\_map’ and other hashing-based containers

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 method 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 in which the keys are stored. 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, and its value can be returned. If it's not present, the key is not stored.

Collisions occur when a computed index position is already occupied by another element. For these situations the abstract containers have solutions available, but that topic is beyond the subject of this chapter.

The Gnu g++ compiler supports the *hash\_(multi)map* and *hash\_(multi)set* containers. Below the *hash\_map* container is discussed, but other containers using hashing (*hash\_multimap*, *hash\_set* and *hash\_multiset*) operate correspondingly.

Concentrating on the *hash\_map*, its constructor needs a *key type*, a *value type*, an object creating a hash value for the key, and an object comparing two keys for equality. Hash functions are available for `char const *` keys, and for all the scalar numerical types `char`, `short`, `int` etc.. If another data type is used, a hash function and an equality test must be implemented, possibly using *function objects* (see section 9.9). For both situations examples are given below.

The class implementing the hash function could be called *hash*. Its function call operator (`operator()()`) returns the hash value of the key that is passed as its argument.

A *generic algorithm* (see chapter 17) exists for the test of equality (i.e., `equal_to()`), which can be used if the key's data type supports the equality operator. Alternatively, a specialized function object could be constructed here, supporting the equality test of two keys. Again, both situations are illustrated below.

The *hash\_map* class implements an associative array in which the key is stored according to some hashing scheme. To use the *hash\_map*, sources must

```
#include <ext/hash_map>
```

The *hash\_(multi)map* is still part of the ANSI/ISO extension. Once this container becomes part of the standard, the `ext/` prefix in the `#include` preprocessor directive can be removed.

Constructors, operators and member functions that are available for the *map* are also available for the *hash\_map*. At the user-level there is no difference in the use of a *map* and a *hash\_map*. However, the *efficiency* of a *hash\_map* in terms of speed should greatly exceed the efficiency of the *map*. Comparable conclusions may be drawn for the *hash\_set*, *hash\_multimap* and the *hash\_multiset*.

Compared to the *map* container, the *hash\_map* has an extra constructor:

```
hash_map<...>
```

```
hash(n);
```

where `n` is an unsigned value, may be used to construct a `hash_map` consisting of an initial number of at least `n` empty slots to put key/value combinations in. This number is automatically extended when needed.

The hashed data type is almost always text. So, a `hash_map` in which the key's data type is either `char const *` or a `string` occurs most often. If the following header file is placed in the C++ compiler's `INCLUDE` path as the file `hashclasses.h`, sources could

```
#include <hashclasses.h>
```

to make available a set of classes that can be used with the instantiation of a hash table. Otherwise, sources must

```
#include <ext/hash_map>

#ifndef _HASHCLASSES_H_
#define _HASHCLASSES_H_

#include <string>
#include <ext/hash_map>

/*
   This file is copyright (c) GPL, 2001.
   =====

   With hash_maps using char const * for the keys:
   =====

   * Use 'HashCharPtr' as 3rd template argument for case-sensitive keys
   * Use 'HashCaseCharPtr' as 3rd template argument for case-insensitive
     keys

   * Use 'EqualCharPtr' as 4th template argument for case-sensitive keys
   * Use 'EqualCaseCharPtr' as 4th template argument for case-insensitive
     keys

   With hash_maps using std::string for the keys:
   =====

   * Use 'HashString' as 3rd template argument for case-sensitive keys
   * Use 'HashCaseString' as 3rd template argument for case-insensitive keys

   * OMIT the 4th template argument for case-sensitive keys
   * Use 'EqualCaseString' as 4th template argument for case-insensitive
     keys

   Examples:

                                   // key is char const *, case sensitive
   hash_map<char const *, int, HashCharPtr, EqualCharPtr >
```

```

        hashtable;

        // key is char const *, case insensitive
        hash_map<char const *, int, HashCaseCharPtr, EqualCaseCharPtr >
        hashtable;

        // key is std::string, case sensitive
        hash_map<std::string, int, HashString>
        hashtable;

        // key is std::string, case insensitive
        hash_map<std::string, int, HashCaseString, EqualCaseString>
        hashtable;

Feb. 2001
Frank B. Brokken (f.b.brokken@rc.rug.nl)
*/

class HashCharPtr
{
public: bool
operator()(char const *str) const
{
    return std::hash<char const *>()(str);
}
};

class EqualCharPtr
{
public: bool
operator()(char const *x, char const *y) const
{
    return !strcmp(x, y);
}
};

class HashCaseCharPtr
{
public: bool
operator()(char const *str) const
{
    std::string
        s = str;
    transform(s.begin(), s.end(), s.begin(), tolower);
    return std::hash<char const *>()(s.c_str());
}
};

class EqualCaseCharPtr
{
public: bool
operator()(char const *x, char const *y) const
{
    return !strcasecmp(x, y);
}
};

```



```

    }
};

class HashString
{
    public: bool
    operator()(std::string const &str) const
    {
        return std::hash<char const *>()(str.c_str());
    }
};

class HashCaseString: public HashCaseCharPtr
{
    public: bool
    operator()(std::string const &str) const
    {
        return HashCaseCharPtr::operator()(str.c_str());
    }
};

class EqualCaseString
{
    public: bool
    operator()(std::string const &s1, std::string const &s2) const
    {
        return !strcasecmp(s1.c_str(), s2.c_str());
    }
};

#endif

```

The following program defines a `hash_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 equality operator used the generic algorithm `equal_to<string>`, which is the default fourth argument of the `hash_map` constructor:

```

//    #include <hashclasses.h>
#include <iostream>
#include "hashclasses.h"

using namespace std;

int main()
{
    hash_map<string, int, HashString >
        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"] << endl <<
        "april      -> " << months["april"] << endl <<
        "june       -> " << months["june"] << endl <<
        "november   -> " << months["november"] << endl;
}
/*
    Generated output:
september -> 30
april      -> 30
june       -> 30
november   -> 30
*/

```

The `hash_multimap`, `hash_set` and `hash_multiset` containers are used analogously. For these containers the `equal` and `hash` classes must also be defined. The `hash_multimap` also requires the `hash_map` header file, the `hash_set` and `hash_multiset` containers can be used when source files

```
#include <ext/hash_set>
```

## 12.3 The ‘complex’ container

The `complex` container is a specialized container in that it defines operations that can be performed on complex numbers, given possible numerical real and imaginary data types.

In order to use the `complex` container, sources must

```
#include <complex>
```

The `complex` container can be used to define complex numbers, consisting of two parts, representing the real and imaginary parts of a complex number.

While initializing (or assigning) a complex variable, the imaginary part may be left out of the initialization or assignment, in which case this part is 0 (zero). By default, both parts are zero.

When complex numbers are defined, the type definition requires the specification of the datatype of the real and imaginary parts. E.g.,

```

complex<double>
complex<int>
complex<float>

```

Note that the real and imaginary parts of complex numbers have the same datatypes.

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 following example two anonymous complex values are pushed on a stack of complex numbers, to be popped again thereafter:

```
#include <iostream>
#include <complex>
#include <stack>

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" << endl;
        cstack.pop();
    }
}
/*
    Generated output:
    -3.14, -2.71i
    3.14, 2.71i
*/
```

Note the required extra blank space between the two closing pointed arrows in the type specification of `cstack`.

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 complex::operator+(value)`:  
this member returns the sum of the current `complex` container and `value`.
  - `complex complex::operator-(value)`:  
this member returns the difference between the current `complex` container and `value`.

- `complex complex::operator*(value):`  
this member returns the product of the current complex container and value.
  - `complex complex::operator/(value):`  
this member returns the quotient of the current complex container and value.
  - `complex complex::operator+=(value):`  
this member adds value to the current complex container, returning the new value.
  - `complex complex::operator-=(value):`  
this member subtracts value from the current complex container, returning the new value.
  - `complex complex::operator*(value):`  
this member multiplies the current complex container by value, returning the new value
  - `complex complex::operator/(value):`  
this member divides the current complex container by value, returning the new value.
- `Type complex::real():`  
this member returns the real part of a complex number.
  - `Type complex::imag():`  
this member 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()`. These functions are normal functions, not member functions. They accept 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., `1.2345`: only the real part, the imaginary part will be set to 0; `(1.2345)`: the same value.

## Chapter 13

# Inheritance

When programming in C, it is common to view problem solutions from a *top-down* 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 dependencies between code and data can also be defined in terms of *classes* which are *related to other classes*. This looks like *composition* (see section 6.4), where objects of a class contain objects of another class as their data. But the relation which is described here is of a different kind: a class can be *defined* by means of an older, pre-existing, class. This leads to a situation in which a new class has all the functionality of the older class, and additionally introduces its own specific functionality. Instead of composition, where a given class *contains* another class, we mean here *derivation*, where a given class *is* 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 definition of the new class. When speaking of 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 will first address the syntactical possibilities which C++ offers to derive classes from other classes. Then we will address some of the possibilities which are thus offered by C++.

As we have seen the object-oriented approach to problem solving in the introductory chapter (see section 2.4), classes are identified during the problem analysis, after which objects of the defined classes can be declared to represent entities of the problem at hand. The classes are placed in a hierarchy, where the top-level class contains the least 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 weight 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.

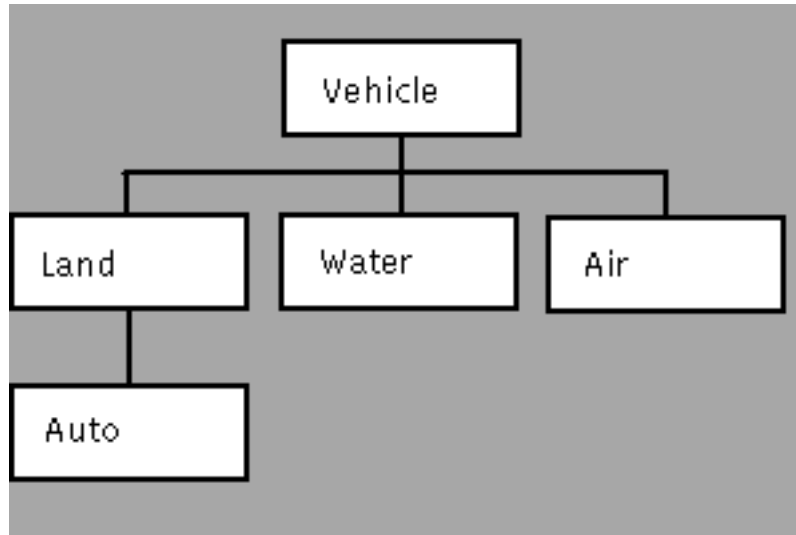


Figure 13.1: Initial object hierarchy of vehicles.

## 13.1 Related types

The relationship between the proposed classes representing different kinds of vehicles is further illustrated here. The figure shows the object hierarchy in vertical direction: an `Auto` is a special case of a `Land` vehicle, which in turn is a special case of a `Vehicle`.

The class `Vehicle` is thus the ‘greatest common denominator’ in the classification system. For the sake of the example we implement in this class the functionality to store and retrieve the weight of a vehicle:

```
class Vehicle
{
    public:
        Vehicle();
        Vehicle(unsigned wt);

        unsigned getweight() const;
        void setweight(unsigned wt);
    private:
        unsigned weight;
};
```

Using this class, the weight of a vehicle can be defined as soon as the corresponding object is created. At a later stage the weight can be re-defined or retrieved.

To represent vehicles which travel over land, a new class `Land` can be defined with the functionality of a `Vehicle`, while adding its own specific information and functionality. Assume that we are interested in the speed of land vehicles *and* in their weights. The relationship between `Vehicles` and `Lands` could of course be represented with composition, but that would be awkward: composition would suggest that a `Land` vehicle *contains* a vehicle, while the relationship should be that the `Land` vehicle *is* a special case of a vehicle.

A relationship in terms of composition would also introduce needless code. E.g., consider the following code fragment which shows a class `Land` using composition (only the `setweight()` func-

tionality is shown):

```
class Land
{
    public:
        void setweight(unsigned wt);
    private:
        Vehicle v;        // composed Vehicle
};

void Land::setweight(unsigned wt)
{
    v.setweight(wt);
}
```

Using composition, the `setweight()` function of the class `Land` only serves to pass its argument to `Vehicle::setweight()`. Thus, as far as weight handling is concerned, `Land::setweight()` introduces no extra functionality, just extra code. Clearly this code duplication is superfluous: a `Land` should *be* a `Vehicle`; it should not *contain* a `Vehicle`.

The intended relationship is better achieved using inheritance: `Land` is *derived* from `Vehicle`, in which `Vehicle` is the base class of the derivation. Here is how such inheritance is achieved:

```
class Land: public Vehicle
{
    public:
        Land();
        Land(unsigned wt, unsigned sp);

        void setspeed(unsigned sp);
        unsigned getspeed() const;

    private:
        unsigned speed;
};
```

By postfixing the class name `Land` in its definition by `: public Vehicle` the derivation is realized: the class `Land` now contains all the functionality of its base class `Vehicle` plus its own specific information and functionality. The extra functionality consists here of a constructor with two arguments and interface functions to access the speed data member.<sup>1</sup> To illustrate the use of the derived class `Land` consider the following example:

```
Land
    veh(1200, 145);

int main()
{
    cout << "Vehicle weighs " << veh.getweight() << endl
         << "Speed is " << veh.getspeed() << endl;
}
```

---

<sup>1</sup> The derivation in this example mentions the keyword `public`: public derivation. C++ also implements private derivation and protected derivation, both of which are not often used and which we will therefore leave to the reader to uncover.

This example shows two features of derivation. First, `getweight()` is no direct member of a `Land`. Nevertheless it is used in `veh.getweight()`. This member function is an implicit part of the class, inherited from its 'parent' vehicle.

Second, although the derived class `Land` now contains the functionality of `Vehicle`, the private fields of `Vehicle` remain private in the sense that they can only be accessed by member functions of `Vehicle` itself. This means that the member functions of `Land` *must* use the interface functions (`getweight()`, `setweight()`) to address the `weight` field; just as any other code outside the `Vehicle` class. This restriction is necessary to enforce the principle of data hiding. The class `Vehicle` could, e.g., be recoded and recompiled, after which the program could be relinked. The class `Land` itself could remain unchanged.

Actually, the previous remark is not quite right: If the internal organization of `Vehicle` changes, then the internal organization of `Land` objects, containing the data of `Vehicle`, changes as well. This means that objects of the `Land` class, after changing `Vehicle`, might require more (or less) memory than before the modification. However, in such a situation we still don't have to worry about the use of member functions of the parent class `Vehicle` in the class `Land`. We might have to recompile the `Land` sources, though, as the relative locations of the data members within the `Land` objects will have changed due to the modification of the `Vehicle` class.

As a rule of thumb, classes which are derived from other classes must be fully recompiled (but don't have to be modified) after changing the *data organization* of their base classes. As adding new member *functions* to the base class doesn't alter the data organization, no recompilation is needed after adding new member *functions*. (A subtle point to note, however, is that adding a new member function that happens to be the *first virtual* member function of a class results in a hidden pointer to a table of pointers to virtual functions. This topic is discussed further in chapter 14).

In the following example we assume that the class `Auto`, representing automobiles, should contain the weight, speed and name of a car. This class is therefore derived from `Land`:

```
class Auto: public Land
{
    public:
        Auto();
        Auto(unsigned wt, unsigned sp, char const *nm);
        Auto(Auto const &other);

        ~Auto();

        Auto const &operator=(Auto const &other);

        char const *getname() const;
        void setname(char const *nm);

    private:
        char const *name;
};
```

In the above class definition, `Auto` is derived from `Land`, which in turn is derived from `Vehicle`. This is called *nested derivation*: `Land` is called `Auto's direct base class`, while `Vehicle` is called the *indirect base class*.

Note the presence of a destructor, a copy constructor and an overloaded assignment operator in the class `Auto`. Since this class uses a pointer to reach dynamically allocated memory, these members should be part of the class interface.



## 13.2 The constructor of a derived class

As mentioned earlier, a derived class inherits the functionality from its base class. In this section we shall describe the effects of the inheritance on the constructor of a derived class.

As will be clear from the definition of the class `Land`, a constructor exists to set both the `weight` and the `speed` of an object. The poor-man's implementation of this constructor could be:

```
Land::Land (unsigned wt, unsigned sp)
{
    setweight(wt);
    setspeed(sp);
}
```

This implementation has the following disadvantage. The C++ compiler will generate code to call the default constructor of a base class from each constructor in the derived class, unless explicitly instructed otherwise. This can be compared to the situation which arises in composed objects (see section 6.4).

Consequently, in the above implementation the default constructor of `Vehicle` is called, which probably initializes the `weight` of the vehicle, only to be redefined immediately thereafter by the function `setweight()`.

A better approach is of course directly to call the constructor of `Vehicle` that expects an `unsigned weight` argument. The syntax to achieve this is to mention the constructor to be called (supplied with an argument) immediately following the argument list of the constructor of the derived class itself. The use of such a base class initializer is shown below:

```
Land::Land(unsigned wt, unsigned sp)
:
    Vehicle(wt)
{
    setspeed(sp);
}
```

## 13.3 The destructor of a derived class

Destructors of classes are automatically called when an object is destroyed. This rule also holds true for objects of classes that are derived from other classes. Assume we have the following situation:

```
class Base
{
    public:
        ~Base();
};

class Derived: public Base
{
    public:
        ~Derived();
};
```

```

int main()
{
    Derived
        derived;
}

```

At the end of the `main()` function, the `derived` object ceases to exist. Hence, its destructor `Derived::~~Derived()` is called. However, since `derived` is also a `Base` object, the `Base::~~Base()` destructor is called as well.

It is this *not* necessary to call the `Base::~~Base()` destructor explicitly from the `Derived::~~Derived()` destructor.

Constructors and destructors are called in a stack-like fashion: when `derived` is constructed, the appropriate `Base` constructor is called first, then the appropriate `Derived` constructor is called. When `derived` is destroyed, the `Derived` destructor is called first, and then the `Base` destructor is called for that object. In general, a derived class destructor is called before a base class destructor is called.

## 13.4 Redefining member functions

The functionality of all members which are defined in a base class (and which are therefore also available in derived classes) can be redefined. This feature is illustrated in this section.

Let's assume that the vehicle classification system should be able to represent trucks, consisting of two parts: the front engine, which pulls a trailer. Both the front engine and the trailer have their own weights, but the `getweight()` function should return the combined weight.

The definition of a `Truck` therefore starts with the class definition, derived from `Auto` but it is then expanded to hold one more unsigned field representing the additional weight information. Here we choose to represent the weight of the front part of the truck in the `Auto` class and to store the weight of the trailer in an additional field:

```

class Truck: public Auto
{
    public:
        Truck();
        Truck(unsigned engine_wt, unsigned sp, char const *nm,
              unsigned trailer_wt);

        void setweight(unsigned engine_wt, unsigned trailer_wt);
        unsigned getweight() const;
    private:
        unsigned trailer_weight;
};

Truck::Truck(unsigned engine_wt, unsigned sp, char const *nm,
             unsigned trailer_wt)
:
    Auto(engine_wt, sp, nm)
{
    trailer_weight = trailer_wt;
}

```

```
}
```

Note that the class `Truck` now contains two functions already present in the base class `Auto`: `setweight()` and `getweight()`.

- The redefinition of `setweight()` poses no problems: this function is simply redefined to perform actions which are specific to a `Truck` object.
- The redefinition of `setweight()`, however, will *hide* `Auto::setweight()`: for a `Truck` only the `setweight()` function having two unsigned arguments can be used.
- The `Vehicle`'s `setweight()` function remains available for a `Truck`, but it *must* now be called *explicitly*, as `Auto::setweight()` is now hidden from view. This latter function is hidden, even though `Auto::setweight()` has only one unsigned argument. To implement `Truck::setweight()` we could write:

```
void Truck::setweight(unsigned engine_wt, unsigned trailer_wt)
{
    trailer_weight = trailer_wt;
    Auto::setweight(engine_wt);    // note: Auto:: is required
}
```

- Outside of the class the `Auto`-version of `setweight()` is accessed through the scope resolution operator. So, if a `Truck t` needs to set its `Auto` weight, it must use

```
t.Auto::setweight(x);
```

- An alternative to using the scope resolution operator is explicitly to include a member function having the same function prototype as the base class member function in the class interface as an inline function. This might be an elegant solution for the occasional situation. E.g., we add the following to the interface of the class `Truck`:

```
void setweight(unsigned engine_wt)
{
    Auto::setweight(engine_wt);
}
```

Now the single argument `setweight()` member function can be used by `Truck` objects without having to use the scope resolution operator. As the function is defined inline, no overhead of an extra function call is involved.

- The function `getweight()` also is already defined in `Auto`, as it was inherited from `Vehicle`. In this case, the class `Truck` should *redefine* this member function to allow for the extra (trailer) weight in the `Truck`:

```
unsigned Truck::getweight() const
{
    return
        (
            Auto::getweight() +    // sum of:
            trailer_weight         //   engine part plus
                                   //   the trailer
        );
}
```

The next example shows the actual use of the member functions of the class `Truck` to display several weights:

```
int main()
{
    Land
        veh(1200, 145);
    Truck
        lorry(3000, 120, "Juggernaut", 2500);

    lorry.Vehicle::setweight(4000);

    cout << endl << "Truck weighs " <<
        lorry.Vehicle::getweight() << endl <<
        "Truck + trailer weighs " <<
        lorry.getweight() << endl <<
        "Speed is " << lorry.getspeed() << endl <<
        "Name is " << lorry.getname() << endl;
}
```

Note the explicit call of `Vehicle::setweight(4000)`: assuming `setweight(unsigned engine_wt)` is not part of the interface of the class `Truck`, it *must* be called explicitly, using the `Vehicle::` scope resolution, as the single argument function `setweight()` is hidden from direct view in the class `Truck`.

The situation with `Vehicle::getweight()` and `Truck::getweight()` is a different one: here the function `Truck::getweight()` is a *redefinition* of `Vehicle::getweight()`, so in order to reach `Vehicle::getweight()` a scope resolution operation (`Vehicle::`) is required.

## 13.5 Multiple inheritance

Up to now, a class was always derived from *one* base class. C++ also allows supports *multiple derivation*, in which a class is derived from several base classes and hence inherits the functionality of multiple parent classes at the same time. In cases where multiple inheritance is considered, it should be defensible to consider the newly derived class an instantiation of both base classes. If that is not really the case, composition might be more appropriate. In general, linear derivation, in which only one base class is used, is seen much more frequently than multiple derivation. Most objects have a primary purpose, and that's it. 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* ....

How can we 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 for the layout of a cockpit instrument panel in an aircraft. 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 first design the `NavSet` class. For the time being, its data members are omitted:

```
class NavSet
{
public:
    NavSet(Intercom &intercom, VHF_Dial &dial);
```

```

        unsigned getActiveFrequency() const;
        unsigned getStandByFrequency() const;

        void setStandByFrequency(unsigned freq);
        unsigned toggleActiveStandby();
        void setVolume(unsigned level);
        void identEmphasis(bool on_off);
};

```

In the class's constructor we assume the availability of the classes `Intercom`, which is used by the pilot to listen to the information that is transmitted through the navigational beacon, and a class `VHF_Dial` which is used to represent visually what the `NavSet` receives.

Next we construct the `ComSet` class. Again, omitting the data members:

```

class ComSet
{
    public:
        ComSet(Intercom &intercom);

        unsigned getFrequency() const;
        unsigned getPassiveFrequency() const;

        void setPassiveFrequency(unsigned freq);
        unsigned toggleFrequencies();

        void setAudioLevel(unsigned level);
        void powerOn(bool on_off);
        void testState(bool on_off);
        void transmit(Message &message);
};

```

In this class we can receive messages, which are transmitted though the `Intercom`, but we can also *transmit* messages, using a `Message` object which is passed to the `ComSet` object using its `transmit()` member function.

Now we're ready to construct the `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 possibilities of either base class are now available in the derived class, using multiple derivation.

With multiple derivation, please note the following:

- The keyword `public` is present before both base class names (`NavSet` and `ComSet`). This is so because the default derivation in C++ is `private`: the keyword `public` must be repeated before each base class specification. The base classes do not have to have the same kind of derivation: one base class could have `public` derivation, another base class could use `protected` derivation, yet 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.

This feature of C++ is frequently used. Usually it pays to develop 'simple' classes each having a simple, well-defined functionality. More complex classes can always be constructed from these simpler building blocks.

- Here is the implementation of The `NavComSet` constructor:

```
NavComSet::NavComSet(Intercom &intercom, VHF_Dial &dial)
:
    ComSet(intercom),
    NavSet(intercom, VHF_Dial)
{}
```

The constructor requires no extra code: Its only 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 code, but by the order in which the base classes are specified in the class interface.

- the `NavComSet` class definition needs no extra 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 bits units in fact use a composed object `Amplifier`, which deals with the volume setting. A revised class might then either use a `Amplifier &getAmplifier()` const member function, and leave it to the application to set up its own interface to the amplifier, or access functions for, e.g., the volume are made available through the `NavSet` and `ComSet` classes as, normally, member functions having the same names (e.g., `setVolume()`). In situations where two base classes use the same member function names, 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 in combination with the doubly occurring member function name:

```
NavComSet
    navcom(intercom, dial);

navcom.NavSet::setVolume(5);    // sets the NavSet volume level
navcom.ComSet::setVolume(5);    // sets the ComSet volume level
```

- The class interface is extended by member functions which do the explicitation for the user of the class. These extra functions will normally be defined as inline:

```
class NavComSet: public ComSet, public NavSet
{
public:
    NavComSet(Intercom &intercom, VHF_Dial &dial);
    void comVolume(unsigned volume)
    {
        ComSet::setVolume(volume);
    }
}
```

```

        void navVolume(unsigned volume)
        {
            NavSet::setVolume(volume);
        }
};

```

- If the NavComSet class is obtained from a third party, and should not be altered, a wrapper class could be used which does the previous explicitation for us in our own programs:

```

class MyNavComSet: public NavComSet
{
public:
    MyNavComSet(Intercom &intercom, VHF_Dial &dial)
    :
        NavComSet(intercom, dial);
    {}
    void comVolume(unsigned volume)
    {
        ComSet::setVolume(volume);
    }
    void navVolume(unsigned volume)
    {
        NavSet::setVolume(volume);
    }
};

```

## 13.6 Conversions between base classes and derived classes

When inheritance is used in the definition of classes, it can be said that an object of a derived class *is* at the same time an object of the base class. This has important consequences for the assignment of objects, and for the situation where pointers or references to such objects are used. Both situations will be discussed next.

### 13.6.1 Conversions in object assignments

Continuing our discussion of the NavCom class, introduced in section 13.5 We start by defining two objects, one of a base class and one of a derived class:

```

ComSet
    com(intercom);
NavComSet
    navcom(intercom2, dial2);

```

The object navcom is constructed using an Intercom and a Dial object. However, a NavComSet is at the same time a ComSet, which makes the assignment *from* navcom (a derived class object) *to* com (a base class object) possible:

```

com = navcom;

```

The effect of this assignment should be that the object `com` will now communicate 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 ignored.

The assignment from a base class object to a derived class object, however, 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.

The following general rule applies: in assignments in which base class objects and derived class objects are involved, assignments in which data are dropped is legal. However, assignments in which data would remain unspecified is *not* allowed. Of course, it is possible to redefine an overloaded assignment operator to allow the assignment of a derived class object by a base class object. E.g., to achieve compilability of a statement

```
navcom = com;
```

the class `NavComSet` must have an overloaded assignment operator function accepting a `ComSet` object for its argument. It would be the programmer's responsibility to decide what to do with the missing data.

### 13.6.2 Conversions in pointer assignments

We return to our `Vehicle` classes, and define the following objects and pointer variable:

```
Land
    land(1200, 130);
Auto
    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:

```
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 which manipulate the `weight` can be called as this is the *only* functionality of a `Vehicle`, which is the object `vp` points to, as far as the compiler can tell.

The same reasoning holds true for references to `Vehicles`. If, e.g., a function is defined with a `Vehicle` reference parameter, the function may be passed an object of a class that is derived from



`Vehicle`. Inside the function, the specific `Vehicle` members of the object of the derived class 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 furthermore has an important consequence for the class `Truck`. After the statement `vp = &truck`, `vp` points to a `Truck` object. So, `vp->getweight()` will return 2600 instead of 8600 (the combined weight of the cabin and of the trailer: 2600 + 6000), which would have been returned by `t.getweight()`.

When a function is called via a pointer to an object, then the *type of the pointer* and not the type of the object itself determines which member functions are available and executed. In other words, C++ implicitly converts the type of an object reached via a pointer to the type of the pointer.

If the actual type of the object to which a pointer points is known, an explicit type cast can be used to access the full set of member functions that are available for the object:

```
Truck
    truck;
Vehicle
    *vp;

vp = &truck;          // vp now points to a truck object

Truck
    *trp;

trp = reinterpret_cast<Truck *>(vp);
cout << "Make: " << trp->getname() << endl;
```

Here, the second to last statement specifically casts a `Vehicle *` variable to a `Truck *`. As is usually the case with type casts, this code is not without risk: it will *only* work if `vp` really points to a `Truck`. Otherwise the program may behave unexpectedly.

## Chapter 14

# Polymorphism

As we have seen in chapter 13, C++ provides the tools to derive classes from base classes, and to use base class pointers to address derived objects. As we've seen, when using a base class pointer to address an object of a derived class, the type of the pointer determines which member function will be used. This means that a `Vehicle *vp`, pointing to a `Truck` object, will incorrectly compute the truck's combined weight in a statement like `vp->getweight()`. The reason for this should now be clear: `vp` calls `Vehicle::getweight()` and not `Truck::getweight()`, even though `vp` actually points to a `Truck`.

Fortunately, a remedy is available. In C++ it is possible for a `Vehicle *vp` to call a function `Truck::getweight()` when the pointer actually points to a `Truck`.

The terminology for this feature is *polymorphism*: it is as though the pointer `vp` changes its type from a base class pointer to a pointer to the class of the object it actually points to. So, `vp` might behave like a `Truck *` when pointing to a `Truck`, and like an `Auto *` when pointing to an `Auto` etc..<sup>1</sup>

Polymorphism is realized by a feature called *late binding*. This refers to the fact that the decision *which* function to call (a base class function or a function of a derived class) cannot be made *compile-time*, but is postponed until the program is actually executed: the actual member function to be used is selected *run-time*.

### 14.1 Virtual functions

The default behavior of the activation of a member function via a pointer or reference is that the type of the pointer (or reference) determines the function that is called. E.g., a `Vehicle *` will activate `Vehicle`'s member functions, even when pointing to an object of a derived class. This is referred to as *early* or *static binding*, since the type of function is known compile-time. The *late* or *dynamic binding* is achieved in C++ using *virtual member functions*.

A member function becomes a virtual member function when its declaration starts with the keyword `virtual`. Once a function is declared `virtual` in a base class, it remains a virtual member function in all derived classes; even when the keyword `virtual` is not repeated in a derived class.

As far as the vehicle classification system is concerned (see section 13.1) the two member functions `getweight()` and `setweight()` might well be declared `virtual`. The relevant sections of the class

---

<sup>1</sup>In 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?"

definitions of the class `Vehicle` and `Truck` are shown below. Also, we show the implementations of the member functions `getweight()` of the two classes:

```
class Vehicle
{
    public:
        virtual int getweight() const;
        virtual void setweight(int wt);
};

class Truck: public Auto
{
    public:
        void setweight(int engine_wt, int trailer_wt);
        int getweight() const;
};

int Vehicle::getweight() const
{
    return (weight);
}

int Truck::getweight() const
{
    return (Auto::getweight() + trailer_wt);
}
```

Note that the keyword `virtual` *only* needs to appear in the definition of the `Vehicle` base class. There is no need (but there is also no penalty) to repeat it in derived classes: once `virtual`, always `virtual`. On the other hand, a function may be declared `virtual` *anywhere* in a class hierarchy: the compiler will be perfectly happy if `getweight()` is declared `virtual` in `Auto`, rather than in `Vehicle`. However, the specific characteristics of virtual member functions would then, for the member function `getweight()`, only appear with `Auto` (and its derived classes) pointers or references. With a `Vehicle` pointer, static binding will remain to be used. The effect of late binding is illustrated below:

```
Vehicle
    v(1200);           // vehicle with weight 1200
Truck
    t(6000, 115,       // truck with cabin weight 6000, speed 115,
      "Scania",        // make Scania, trailer weight 15000
      15000);
Vehicle
    *vp;               // generic vehicle pointer

int main()
{
    vp = &v;           // see (1) below
    cout << vp->getweight() << endl;

    vp = &t;           // see (2) below
    cout << vp->getweight() << endl;

    cout << vp->getspeed() << endl;    // see (3) below
}
```

Since the function `getweight()` is defined `virtual`, late binding is used:

- at (1), `Vehicle::getweight()` is called.
- at (2) `Truck::getweight()` is called.
- at (3) a syntax error is generated. The member `getspeed()` is no member of `Vehicle`, and hence not callable via a `Vehicle*`.

The example illustrates that when using a pointer to a class, *only the functions which are members of that class can be called*. These functions *may* be `virtual`, but this only influences the type of binding (early vs. late).

A virtual member function cannot be a static member function: a virtual member function is still a ordinary member function in that it has a `this` pointer. As static member functions have no `this` pointer, they cannot be declared `virtual`.

## 14.2 Virtual destructors

When the operator `delete` releases memory which is occupied by a dynamically allocated object, or when an object goes out of scope, the appropriate destructor is called to ensure that memory allocated by the object is also deleted. Now consider the following code fragment (cf. section 13.1):

```
Vehicle
    *vp = new Land(1000, 120);

delete vp;           // object destroyed
```

In this example an object of a derived class (`Land`) is destroyed using a base class pointer (`Vehicle *`). For a 'standard' class definition this will mean that the destructor of `Vehicle` is called, instead of the destructor of the `Land` object. This not only results in a memory leak when memory is allocated in `Land`, but it will also prevent any other task, normally performed by the derived class' destructor from being completed (or, better: started). A Bad Thing.

In C++ this problem is solved using a *virtual destructor*. By applying the keyword `virtual` to the declaration of a destructor the appropriate derived class destructor is activated when the argument of the `delete` operator is a base class pointer. In the following partial class definition the declaration of such a virtual destructor is shown:

```
class Vehicle
{
    public:
        virtual ~Vehicle();
        virtual unsigned getweight() const;
};
```

By declaring a virtual destructor, the above `delete` operation (`delete vp`) will correctly call the `Land`'s destructor, rather than the `Vehicle`'s destructor.

From this discussion we are now able to formulate the following situations in which a destructor should be defined:

- A destructor should be defined when memory is allocated and managed by objects of the class.

- A virtual destructor should be defined if the class contains at least one virtual member function.

In the second case, the destructor will have no special tasks to perform. The virtual destructor will therefore often be defined empty. For example, the definition of `Vehicle::~~Vehicle()` may be as simple as:

```
Vehicle::~~Vehicle()
{}
```

Often this will be part of the class interface as an inline destructor.

## 14.3 Pure virtual functions

Until now the base class `Vehicle` contained its own, concrete, implementations of the virtual functions `getweight()` and `setweight()`. In C++ it is however also possible only to *mention* virtual member functions in a base class, without actually defining them. The functions are concretely implemented in a derived class. This approach defines a *protocol*, which has to be followed in the derived classes. This implies that derived classes must take care of the actual definition: the C++ compiler will not allow the definition of an object of a class in which one or more member functions are left undefined. The base class thus enforces a protocol by declaring a function by its name, return value and arguments. The derived classes must take care of the actual implementation. The base class itself defines therefore only a *model* or *mold*, to be used when other classes are derived. Such base classes are also called *abstract classes*.

The functions which are only declared in the base class are called *pure virtual functions*. A function is made pure virtual by preceding its declaration with the keyword `virtual` and by postfixing it with `= 0`. An example of a pure virtual function occurs in the following listing, where the definition of a class `Object` requires the implementation of the conversion operator `operator string()`:

```
#include <string>

class Object
{
public:
    virtual operator string() const = 0;
};
```

Now, all classes derived from `Object` *must* implement the `operator string()` member function, or their objects cannot be constructed. This is neat: all objects derived from `Object` can now always be considered string objects, so they can, e.g., be inserted into ostream objects.

Should the virtual destructor of a base class be a pure virtual function? The answer to this question is no: a class such as `Vehicle` should not *require* derived classes to define a destructor. In contrast, `Object::operator string()` can be a pure virtual function: in this case the base class defines a protocol which must be adhered to.

Realize what would happen if we would define the destructor of a base class as a pure virtual destructor: according to the *compiler*, the derived class object can be constructed: as its destructor is defined, the derived class is not a pure abstract class. However, inside the derived class destructor, the destructor of its base class is implicitly called. This destructor was never defined, and the *linker* will loudly complain about an undefined reference to, e.g., `Virtual::~~Virtual()`.

Often, but not necessarily always, 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 applies also to pure virtual functions: if the member function will alter the object's data members, it cannot be a `const` member function. Often abstract base classes have no data members. However, the prototype of the pure virtual member function must be used again in derived classes. If the implementation of a pure virtual function in a derived class alters the data of the derived class object, then *that* function cannot be declared as a `const` member function. Therefore, the constructor of an abstract base class should well consider whether a pure virtual member function should be a `const` member function or not.

## 14.4 Virtual functions in multiple inheritance

As mentioned in chapter 13 it is possible to derive a class from several base classes. Such a derived class inherits the properties of all its base classes. Of course, the base classes themselves may be derived from classes yet higher in the hierarchy.

A slight difficulty in multiple inheritance may arise when more than one 'path' leads from the derived class to the base class. This is illustrated in the code example below: a class `Derived` is doubly derived from a class `Base`:

```
class Base
{
    public:
        void setfield(int val)
            { field = val; }
        int getfield() const
            { return (field); }
    private:
        int field;
};

class Derived: public Base, public Base
{
};
```

Due to the double derivation, the functionality of `Base` now occurs twice in `Derived`. This leads to ambiguity: when the function `setfield()` is called for a `Derived` object, *which* function should that be, since there are two? In such a duplicate derivation, many C++ compilers will refuse to generate code and will (correctly) identify an error.

The above code clearly duplicates its base class in the derivation. Such a duplication can here easily be easily. But duplication of a base class can also occur through nested inheritance, where an object is derived from, say, an `Auto` and from an `Air` (see the vehicle classification system, section 13.1). Such a class would be needed to represent, e.g., a flying car<sup>2</sup>. An `AirAuto` would ultimately contain two `Vehicles`, and hence two weight fields, two `setweight()` functions and two `getweight()` functions.

---

<sup>2</sup>such as the one in James Bond vs. the Man with the Golden Gun...

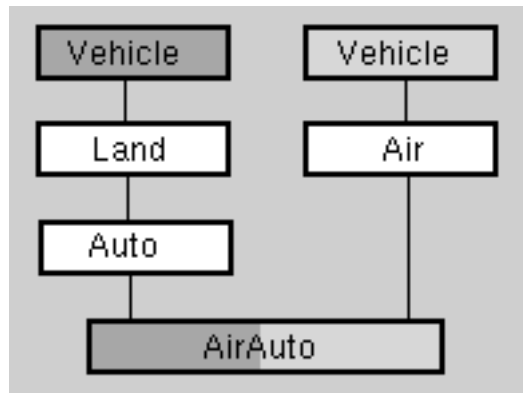


Figure 14.1: Duplication of a base class in multiple derivation.

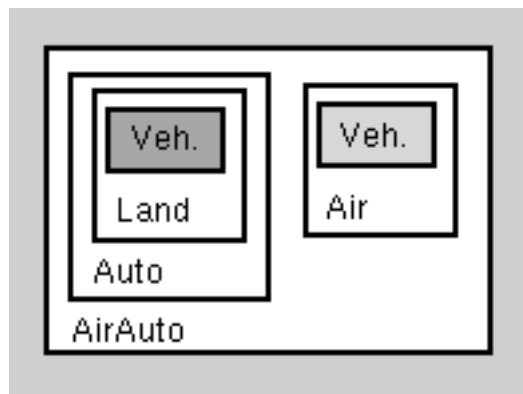


Figure 14.2: Internal organization of an AirAuto object.

#### 14.4.1 Ambiguity in multiple inheritance

Let's investigate closer why an `AirAuto` introduces ambiguity, when derived from `Auto` and `Air`.

- An `AirAuto` is an `Auto`, hence a `Land`, and hence a `Vehicle`.
- However, an `AirAuto` 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 `AirAuto` is shown in figure 14.2

The C++ compiler will detect the ambiguity in an `AirAuto` object, and will therefore fail to compile a statement like:

```
AirAuto
    cool;

cout << cool.getweight() << endl;
```

The question of which member function `getweight()` should be called, cannot be answered by the compiler. The programmer has two possibilities to resolve the ambiguity explicitly:

- First, the function call where the ambiguity occurs can be modified. This is done with the scope resolution operator:

```
// let's hope that the weight is kept in the Auto
// part of the object..
cout << cool.Auto::getweight() << endl;
```

Note the position of the scope operator and the class name: before the name of the member function itself.

- Second, a dedicated function `getweight()` could be created for the class `AirAuto`:

```
int AirAuto::getweight() const
{
    return(Auto::getweight());
}
```

The second possibility from the two above is preferable, since it relieves the programmer who uses the class `AirAuto` of special precautions.

However, apart from these explicit solutions, there is a more elegant one, which will be introduced in the next section.

### 14.4.2 Virtual base classes

As illustrated in figure 14.2, more than one object of the class `Vehicle` is present in an `AirAuto`. The result is not only an ambiguity in the functions which access the `weight` data, but also the presence of two `weight` fields. This is somewhat redundant, since we can assume that an `AirAuto` has just one `weight`.

We can achieve the situation that only one `Vehicle` will be contained in an `AirAuto`. This is done by ensuring that the base class which is multiply present in a derived class, is defined as a *virtual base class*. For the class `AirAuto` this means that the derivation of `Land` and `Air` is changed:

```
class Land: virtual public Vehicle
{
    // etc
};

class Air: virtual public Vehicle
{
    // etc
};

class AirAuto: public Land, public Air
{
};
```

The virtual derivation ensures that via the `Land` route, a `Vehicle` is only added to a class when a virtual base class was not yet present. The same holds true for the `Air` route. This means that we can no longer say via which route a `Vehicle` becomes a part of an `AirAuto`; we can only say that there is an embedded `Vehicle` object. The internal organization of an `AirAuto` after virtual derivation is shown in figure 14.3.



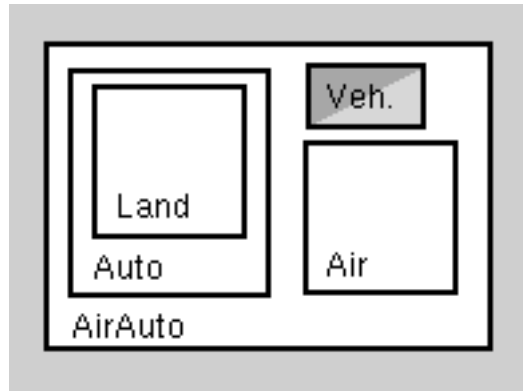


Figure 14.3: Internal organization of an AirAuto object when the base classes are virtual.

There are several points worth noting when using virtual derivation:

- When base classes of a class using multiple derivation are themselves virtually derived from a base class (as shown above), the base class constructor which is normally called when the derived class constructor is called is no longer called: its base class initializer is *ignored*. Instead, the base class constructor will be called independently from the derived class constructors. Assume we have two classes, `Derived1` and `Derived2`, both (possibly virtually) derived from `Base`. We will address the question which constructors will be called when a class `Final: public Derived1, public Derived2` is defined. To distinguish the several constructors that are involved, we will use `Base1()` to indicate the `Base` class constructor that is called as base class initializer for `Derived1` (and analogously: `Base2()` belonging to `Derived2`), while `Base()` indicates the default constructor of the class `Base`. Apart from the `Base` class constructor, we use `Derived1()` and `Derived2()` to indicate the base class initializers for the class `Final`. We now distinguish the following situation, for constructors of the class `Final: public Derived1, public Derived2`:

– classes: `Derived1: public Base, Derived2: public Base`

This is the normal, non virtual multiple derivation. There are two `Base` classes in the `Final` object, and the following constructors will be called (in the mentioned order):

\* `Base1()`, `Derived1()`, `Base2()`, `Derived2()`

– classes: `Derived1: public Base, Derived2: virtual public Base`

Only `Derived2` uses virtual derivation. For the `Derived2` part the base class initializer will be omitted, and the default `Base` class constructor will be called. Furthermore, this ‘detached’ base class constructor will be called *first*:

\* `Base()`, `Base1()`, `Derived1()`, `Derived2()`

Note that `Base()` is called first, *not* `Base1()`. Also note that, as only one derived class uses virtual derivation, there are still *two* `Base` class objects in the eventual `Final` class. Merging of base classes only occurs with multiple virtual base classes.

– classes: `Derived1: virtual public Base, Derived2: public Base`

Only `Derived1` uses virtual derivation. For the `Derived1` part the base class initializer will now be omitted, and the default `Base` class constructor will be called instead. Note the difference with the first case: `Base1()` is replaced by `Base()`. Should `Derived1` happen to use the default `Base` constructor, no difference would be noted here with the first case:

\* `Base()`, `Derived1()`, `Base2()`, `Derived2()`

– classes: `Derived1: virtual public Base, Derived2: virtual public Base`

Here both derived classes use virtual derivation, and so only *one* Base class object will be present in the Final class. Note that now only one Base class constructor is called: for the detached (merged) Base class object:

`* Base(), Derived1(), Derived2()`

- Virtual derivation is, in contrast to virtual functions, a pure compile-time issue: whether a derivation is virtual or not defines how the compiler builds a class definition from other classes.

Summarizing, using virtual derivation avoids ambiguity in the calling of member functions of a base class. Furthermore, duplication of data members is avoided.

### 14.4.3 When virtual derivation is not appropriate

In contrast to the previous definition of a class such as `AirAuto`, situations may arise where the double presence of the members of a base class is appropriate. To illustrate this, consider the definition of a `Truck` from section 13.4:

```
class Truck: public Auto
{
    public:
        Truck();
        Truck(int engine_wt, int sp, char const *nm,
              int trailer_wt);

        void setweight(int engine_wt, int trailer_wt);
        int getweight() const;
    private:
        int trailer_weight;
};

Truck::Truck(int engine_wt, int sp, char const *nm,
             int trailer_wt)
:
    Auto(engine_wt, sp, nm)
{
    trailer_weight = trailer_wt;
}

int Truck::getweight() const
{
    return
    (
        Auto::getweight() +    // sum of:
        trailer_wt             // engine part plus
                               // the trailer
    );
}
```

This definition shows how a `Truck` object is constructed to contain two weight fields: one via its derivation from `Auto` and one via its own `int trailer_weight` data member. Such a definition is of course valid, but it could also be rewritten. We could derive a `Truck` from an `Auto` *and* from a

Vehicle, thereby explicitly requesting the double presence of a Vehicle; one for the weight of the engine and cabin, and one for the weight of the trailer. A small point of interest here is that a derivation like

```
class Truck: public Auto, public Vehicle
```

is not accepted by the C++ compiler: a Vehicle is already part of an Auto, and is therefore not needed. An intermediate class solves the problem: we derive a class TrailerVeh from Vehicle, and Truck from Auto and from TrailerVeh. All ambiguities concerning the member functions are then be solved for the class Truck:

```
class TrailerVeh: public Vehicle
{
    public:
        TrailerVeh(int wt);
};

TrailerVeh::TrailerVeh(int wt)
:
    Vehicle(wt)
{
}

class Truck: public Auto, public TrailerVeh
{
    public:
        Truck();
        Truck(int engine_wt, int sp, char const *nm,
              int trailer_wt);

        void setweight(int engine_wt, int trailer_wt);
        int getweight() const;
};

Truck::Truck(int engine_wt, int sp, char const *nm,
             int trailer_wt)
:
    Auto(engine_wt, sp, nm),
    TrailerVeh(trailer_wt)
{
}

int Truck::getweight() const
{
    return
        (
            Auto::getweight() +           // sum of:
            TrailerVeh::getweight()       //   engine part plus
        );                               //   the trailer
}
```

## 14.5 Run-Time Type identification

C++ offers two ways to retrieve the type of objects and expressions while the program is running. The possibilities of C++'s *run-time type identification* are somewhat limited compared to languages like **Java**. Normally, C++ uses static type checking and static type identification. Static type checking and determination is possibly safer and certainly more efficient than run-time type identification, and should therefore be used wherever possible. Nonetheless, C++ offers run-time type identification by providing the *dynamic cast* and *typeid* operators.

- The `dynamic_cast<>()` operator can be used to convert a base class pointer or reference to a derived class pointer or reference.
- The `typeid` operator returns the actual type of an expression.

These operators operate on class type objects, containing at least one virtual member function.

### 14.5.1 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.

A dynamic cast is performed run-time. A prerequisite for the use of the dynamic cast operator is the existence of at least one virtual member function in the base class.

In the following example a pointer to the class `Derived` is obtained from the `Base` class pointer `bp`:

```
class Base
{
    public:
        virtual ~Base();
};

class Derived: public Base
{
    public:
        char const *toString()
        {
            return ("Derived object");
        }
};

int main()
{
    Base
        *bp;
    Derived
        *dp,
        d;

    bp = &d;

    dp = dynamic_cast<Derived *>(bp);
```

```

    if (dp)
        cout << dp->toString() << endl;
    else
        cout << "dynamic cast conversion failed\n";
}

```

Note the test: in the if condition the success of the dynamic cast is checked. This must be done *run-time*, as the compiler can't do this all by itself. 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. Consequently, if there are multiple derived classes, a series of checks could be performed to find the actual derived class to which the pointer points (In the next example derived classes are declared only):

```

class Base
{
    public:
        virtual ~Base();
};
class Derived1: public Base;
class Derived2: public Base;

int main()
{
    Base
        *bp;
    Derived1
        *d1,
        d;
    Derived2
        *d2;

    bp = &d;

    if ((d1 = dynamic_cast<Derived1 *>(bp)))
        cout << *d1 << endl;
    else if ((d2 = dynamic_cast<Derived2 *>(bp)))
        cout << *d2 << endl;
}

```

Alternatively, a reference to a base class object may be available. In this case the `dynamic_cast<>()` operator will throw an exception if it fails. For example:

```

#include <iostream>
#include <typeinfo>

class Base
{
    public:
        virtual ~Base()
        {}
        virtual char const *toString()
        {}
};

```

```

class Derived1: public Base
{
};

class Derived2: public Base
{
};

void process(Base &b)
{
    try
    {
        cout << dynamic_cast<Derived1 &>(b).toString() << endl;
        return;
    }
    catch (std::bad_cast)
    {}

    try
    {
        cout << dynamic_cast<Derived2 &>(b).toString() << endl;
        return;
    }
    catch (std::bad_cast)
    {}
}

int main()
{
    Derived1
        d;

    process(d);
}

```

In this example the value `std::bad_cast` is introduced. The `std::bad_cast` is thrown as an exception if the dynamic cast of a reference to a base class object fails. Apparently `bad_cast` is the name of a type (). In section `EMPTYENUM` the construction of such a type is discussed.

The dynamic cast operator is a handy 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 be able to use the derived class' functionality.

Casts from a base class reference or pointer to a derived class reference or pointer are called *down-casts*.

### 14.5.2 The typeid operator

As with the `dynamic_cast<>()` operator, the `typeid` is usually applied to base class objects, that are actually derived class objects. Similarly, the base class should contain one or more virtual functions.

In order to use the `typeid` operator, source files must

```
#include <typeinfo>
```

Actually, the typeid operator returns an object of type `type_info`, which may, e.g., be compared to other `type_info` objects.

The class `type_info` may be implemented differently by different implementations, but at the very least it has the following interface:

```
class type_info
{
public:
    virtual ~type_info();
    int operator==(const type_info &other) const;
    int operator!=(const type_info &other) 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 overloaded assignment operator. This prevents the normal construction or assignment of a `type_info` object. `type_info` objects are constructed and returned by the typeid operator. Implementations, however, may choose to extend or elaborate the `type_info` class and provide, e.g., lists of functions that can be called with a certain class.

If the typeid operator is given a base class reference (where the base class contains at least one virtual function), it will indicate that the type of its operand is the derived class. For example:

```
class Base;    // contains >= 1 virtual functions
class Derived: public Base;

Derived
    d;
Base
    &br = d;

cout << typeid(br).name() << endl;
```

In this example the typeid operator is given a base class reference. It will print the text “Derived”, being the class name of the class `br` actually refers to. If `Base` does not contain virtual functions, the text “Base” would have been 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:

```
cout << typeid(12).name() << endl;    // prints:  int
cout << typeid(12.23).name() << endl; // prints:  double
```

Note, however, that the above example is suggestive at most of the type that is printed. It *may* be `int` and `double`, but this is not necessarily the case. If portability is required, make sure no tests against static, built-in 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, it is important to realize that a base class *reference* is used as the argument of the `typeid` operator. Consider the following example:

```
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
    ...
```

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 with

```
if (typeid(*bp) == typeid(Derived))    // 3: true
    ...
if (typeid(*bp) == typeid(Base))       // 4: false
    ...
```

we see that (3) now returns true: `*bp` actually refers to a `Derived` class object, and `typeid(*bp)` will return `typeid(Derived)`.

A similar result is obtained if a base class reference is used:

```
Base
    &br = *bp;

if (typeid(br) == typeid(Derived))     // 3: true
    ...
if (typeid(br) == typeid(Base))        // 4: false
    ...
```

## 14.6 Deriving classes from ‘streambuf’

The class `streambuf` (see section 5.7 and figure 5.2) has many (protected) virtual member functions (see section 5.7.1) that are used by the `stream` classes using `streambuf` objects. By deriving a class from the class `streambuf` these member functions may be overridden in the derived classes, thus implementing a specialization of the class `streambuf` for which the standard `istream` and `ostream` objects can be used.



Basically, a `streambuf` interfaces to some *device*. The normal behavior of the `stream`-class objects remains unaltered. So, a string extraction from a `streambuf` object will still return a consecutive sequence of non white space delimited characters. If the derived class is used for *input operations*, the following member functions are serious candidates to be overridden. Examples in which some of these functions are overridden will be given later in this section:

- `int streambuf::pbackfail(int c):`

This member 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 == eofFile()` then the input device must be reset one character, otherwise `c` must be prepended to the characters to be read. The function returns `EOF` on failure. Otherwise 0 can be returned. The function is called when other attempts to push back a character fail.

- `streamsize streambuf::showmanyc():`

This member 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 at least 0 characters will be returned before the latter two functions will return `EOF`). When a positive value is returned then the next call to the `underflow()` member will not return `EOF`.

- `int streambuf::uflow():`

By default, this function calls `underflow()`. If `underflow()` fails, `EOF` is returned. Otherwise, the next character available character is returned as `*gptr()` following a `gbump(-1)`. The member moves the pending character that is returned also to the backup sequence. This is different from `underflow()`, which also returns the next available character, but does not alter the input position.

- `int streambuf::underflow():`

This member is called when

- there is no input buffer (`eback() == 0`)
- `gptr() >= egptr()`: there are no more pending input characters.

It returns the next available input character, which is the character at `gptr()`, or the first available character from the input device.

Since this member is eventually used by other member functions for reading characters from a device, at the very least this member function must be overridden for new classes derived from `streambuf`.

- `streamsize streambuf::xsgetn(char *buffer, streamsize n):`

This member function should act as if the returnvalues of `n` calls of `snext()` are assigned to consecutive locations of `buffer`. If `EOF` is returned then reading stops. The actual number of characters read is returned. Overridden versions could optimize the reading process by, e.g., directly accessing the input buffer.

When the derived class is used for *output operations*, the next member functions should be considered:

- `int streambuf::overflow(int c):`

This member is called to write characters from the pending sequence to the output device. Unless `c` is EOF, it is at least logically, appended to the pending sequence. So, if the pending sequence consists of the characters 'h', 'e', 'l' and 'l', and `c == 'o'`, then eventually 'hello' will be written to the output device.

Since this member is eventually used by other member functions for writing characters to a device, at the very least this member function must be overridden for new classes derived from `streambuf`.

- `streamsize streambuf::xsputn(char const *buffer, streamsize n):`

This member function should act as if `n` consecutive locations of `buffer` are passed to `sputc()`. If EOF is returned by this latter member, then writing stops. The actual number of characters written is returned. Overridden versions could optimize the writing process by, e.g., directly accessing the output buffer.

For derived classes using buffers and supporting seek operations, consider these member functions:

- `streambuf *streambuf::setbuf(char *buffer, streamsize n):`

This member function is called by the `pubsetbuf()` member function.

- `pos_type streambuf::seekoff(off_type offset, ios::seekdir way, ios::openmode mode = ios::in | ios::out):`

This member function is called to reset the position of the next character to be processed. It is called by `pubseekoff()`. The new position or an invalid position (e.g., -1) is returned.

- `pos_type streambuf::seekpos(pos_type offset, ios::openmode mode = ios::in | ios::out):`

This member function acts similarly as `seekoff()`, but operates with absolute rather than relative positions.

- `int sync():`

This member function flushes all pending characters to the device, and/or resets an input device to the position of the first pending character, waiting in the input buffer to be consumed. It returns 0 on success, -1 on failure. As the default `streambuf` is not buffered, the default implementation also returns 0.

Next, consider the following problem, which will be solved by constructing a class `capsbuf` that is derived from `streambuf`. The problem is to construct a `streambuf` which writes its information to the standard output stream in such a way that all white-space series of characters are capitalized. The class `capsbuf` obviously needs an overridden `overflow()` member and a minimal awareness of its state. Its state changes from 'Capitalize' to 'Literal' as follows:

- The start state is 'Capitalize';
- Change to 'Capitalize' after processing a white-space character;
- Change to 'Literal' after processing a non-whitespace character.

A simple variable to remember the last character allows us to keep track of the current state. Since 'Capitalize' is similar to 'last character processed is a white space character' we can simply initialize

the variable with a white space character, e.g., the blank space. Here is the initial definition of the class capsbuf:

```
#include <iostream>
#include <streambuf>
#include <ctype.h>

class capsbuf: public std::streambuf
{
    public:
        capsbuf()
        :
            last(' ')
        {}

    protected:
        int overflow(int c)          // interface to the device.
        {
            std::cout.put(isspace(last) ? toupper(c) : c);
            last = c;
            return c;
        }
    private:
        int last;
};
```

An example of a program using capsbuf is:

```
#include "capsbuf1.h"

using namespace std;

int main()
{
    capsbuf
        cb;

    ostream
        out(&cb);

    out << hex << "hello " << 32 << " worlds" << endl;
}
/*
    Generated output:
    Hello 20 Worlds
*/
```

Note the use of the insertion operator, and note that all type and radix conversions (inserting hex and the value 32, coming out as the ASCII-characters '2' and '0') is neatly done by the ostream object. The real purpose in life for capsbuf is to capitalize series of ASCII-characters, and that's what it does very well.

Next, we realize that inserting characters into streams can also be done using a construction like

```
cout << cin.rdbuf();
```

or, boiling down to the same thing:

```
cin >> cout.rdbuf();
```

Realizing that this is all about streams, we now try, in the above `main()` function:

```
cin >> out.rdbuf();
```

We compile and link the program to the executable `caps`, and start:

```
echo hello world | caps
```

Unfortunately, nothing happens.... Any reaction is also lacking if we try the statement `cin >> cout.rdbuf()`. What's wrong here?

The difference between `cout << cin.rdbuf()`, which *does* produce the expected results and our using of `cin >> out.rdbuf()` is that the operator `>>(streambuf *)` (and its insertion counterpart) member function only does a `streambuf`-to-`streambuf` copy if the respective stream modes are set up correctly. So, the argument of the extraction operator must point to a `streambuf` into which information can be written. By default, no stream mode is set for a plain `streambuf` object. As there is no constructor for a `streambuf` accepting an `ios::openmode`, we force the `ios::out` mode by defining an output buffer using `setp()`. By doing so we define a buffer, but don't want to use it, so we make its size 0. Note that this is something different than using 0-argument values with `setp()`, as this would indicate 'no buffering', which would not alter the default situation. Although any non-0 value could be used for the empty `[begin, begin)` range, we decided to define a (dummy) local `char` variable in the constructor, and use  `[&dummy, &dummy)` to define the empty buffer. This effectively makes `capsbuf` an output buffer, thus activating the

```
istream::operator>>(streambuf *)
```

member. Here is the revised constructor of the class `capsbuf`:

```
capsbuf::capsbuf()
:
    last(' ')
{
    char
        dummy;
    setp(&dummy, &dummy);
}
```

Now the program can use either

```
out << cin.rdbuf();
```

or:

```
cin >> out.rdbuf();
```

Actually, the `ostream` wrapper isn't really needed here:

```
cin >> &cb;
```

would have produced the same results.

It is not clear whether the `setp()` solution proposed here is actually a *kludge*. After all, shouldn't the `ostream` wrapper around `cb` inform the `capsbuf` that it should act as a `streambuf` for doing output operations?

## 14.7 A polymorphic exception class

Earlier in the Annotations (section 8.3.1) 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 will now probably be clear that our class `Exception` should be a virtual base class, from which special exception handling classes can be derived. It could even be argued that `Exception` can be an abstract base class declaring only pure virtual member functions. In the discussion in section 8.3.1 a member function `severity()` was mentioned which might not be a proper candidate for a purely abstract member function, but for that member we can now use the completely general `dynamic_cast<>()` operator.

The (abstract) base class `Exception` is designed as follows:

```
#ifndef _EXCEPTION_H_
#define _EXCEPTION_H_

#include <iostream>
#include <string>

class Exception
{
    friend ostream &operator<<(ostream &str, Exception const &e)
    {
        return str << e.operator string();
    }

public:
    virtual ~Exception()
    {}
    virtual void process() const = 0;
    virtual operator string() const
    {
        return reason;
    }
protected:
    Exception(char const *reason)
    :
        reason(reason)
    {}
    string
```

```

        reason;
};
#endif

```

The operator `string()` member function of course replaces the `toString()` member used in section 8.3.1. The friend operator `<<()` function is using the (virtual) operator `string()` member so that we're able to insert an `Exception` object into an ostream. Apart from that, notice the use of a virtual destructor, doing nothing.

A derived class `FatalException`: `public Exception` could now be defined as follows (using a very basic `process()` implementation indeed):

```

#ifndef _FATALEXCEPTION_H_
#define _FATALEXCEPTION_H_

#include "exception.h"

class FatalException: public Exception
{
public:
    FatalException(char const *reason)
    :
        Exception(reason)
    {}
    void process() const
    {
        exit(1);
    }
};
#endif

```

The translation of the example at the end of section 8.3.1 to the current situation can now easily be made (using derived classes `WarningException` and `MessageException`), constructed like `FatalException`:

```

#include <iostream>

#include "message.h"
#include "warning.h"

void initialExceptionHandler(Exception const *e)
{
    cout << *e << endl;           // show the plain-text information

    if
    (
        dynamic_cast<MessageException const *>(e)
        ||
        dynamic_cast<WarningException const *>(e)
    )
    {
        e->process();               // Process a message or a warning
        delete e;
    }
    else

```

```

        throw;                                // Pass on other types of Exceptions
    }

```

## 14.8 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 using this feature is the only intention. However, we think it's nice to know how polymorphism is at all possible. Besides, the following discussion does explain why there is a cost of polymorphism in terms of memory usage.

The fundamental idea behind polymorphism is that the compiler does not know which function to call compile-time; the appropriate function will be selected run-time. That means that the address of the function must be stored somewhere, to be looked up prior to the actual call. This 'somewhere' place must be accessible from the object in question. E.g., when a `Vehicle *vp` points to a `Truck` object, then `vp->getweight()` calls a member function of `Truck`; the address of this function is determined from the actual object which `vp` points to.

A common implementation is the following: An object containing virtual member functions holds as its first data member a hidden field, pointing to an array of pointers containing the addresses of the virtual member functions. The hidden data member is usually called the *vp pointer*, the array of virtual member function addresses the *vtable*. Note that the discussed implementation is compiler-dependent, and is by no means dictated by the C++ ANSI/ISO standard.

The table of addresses of virtual functions is shared by all objects of the class. Multiple classes may even share the same table. The overhead in terms of memory consumption is therefore:

- One extra pointer field per object, which points to:
- One table of pointers per (derived) class to address the virtual functions.

Consequently, a statement like `vp->getweight()` 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 of two addresses: one pointer for the function `getweight()` and one pointer for the function `setweight()`. The actual function which is called is determined from this table.

The internal organization of the objects having virtual functions is further illustrated in figure 14.4.

As can be seen from figure 14.4, all objects which use virtual functions must have one (hidden) data member to address a table of function pointers. The objects of the classes `Vehicle` and `Auto` both address the same table. The class `Truck`, however, introduces its own version of `getweight()`: therefore, this class needs its own table of function pointers.

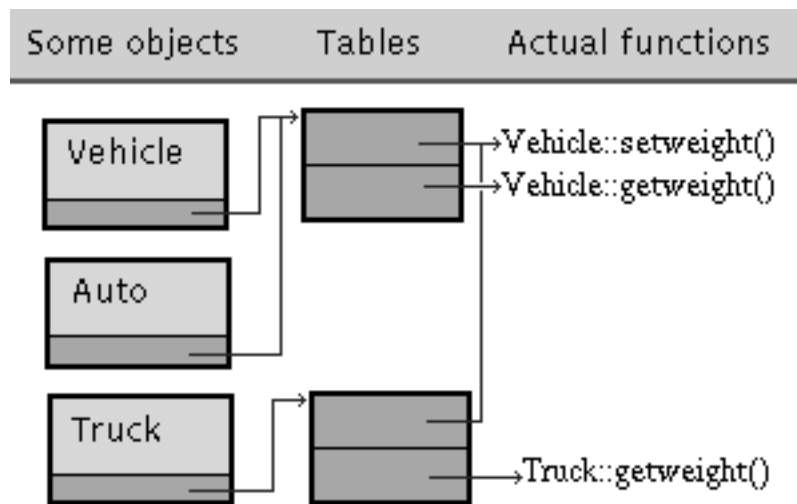


Figure 14.4: Internal organization objects when virtual functions are defined.



## Chapter 15

# Classes having pointers to members

Classes having pointer data members have been discussed in detail in chapter 6. As we have seen, when pointer data-members occur in classes, such classes deserve some special treatment.

By now it is well known how to treat pointer data members: constructors are used to initialize pointers, destructors are needed to delete the memory pointed to by the pointer data members.

Furthermore, in classes having pointer data members copy constructors and overloaded assignment operators are normally needed as well.

However, in some situations we do not need a pointer to an object, but rather a pointer to members of an object. In this chapter these special pointers are the topic of discussion.

### 15.1 Pointers to members: an example

Knowing how pointers to variables and objects are used does not intuitively lead to the concept of a *pointer to members*. Even if the return type and parameter types of a member function are taken into account, surprises are likely to be encountered. For example, consider the following class:

```
class String
{
    public:
        char const *get() const;
    private:
        char const *(*sp)() const;
};
```

For this class, it is not possible to define a `char const *(*sp)() const` pointing to the `get()` member function of the `String` class: `sp` cannot be given the address of the member function `get()`.

One of the reasons why this doesn't work is that the variable `sp` has a global scope, while the member function `get()` is defined within the `String` class, and has `class` scope. The fact that the variable `sp` is part of the `String` class is of no relevance. According to `sp`'s definition, it points to a function living *outside* of the class.

Consequently, in order to define a pointer to a member (either data or function, but usually a function) of a class, the scope of the pointer must be within the class' scope. Doing so, a pointer

to a member of the class `String` can be defined as

```
char const
    *(String::*sp)() const;
```

So, due to the `String::` prefix, `sp` is defined to be active only in the context of the class `String`. Here, it is defined as a pointer to a constant function, not expecting arguments, and returning a pointer to constant characters.

## 15.2 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 `char const * (String::*sp)() const` to indicate:

- `sp` is a pointer (`*sp`),
- to something in the class `String` (`String::*sp`).
- It is a pointer to a `const` function, returning a `char const *`: `char const * (String::*sp)() const`
- The prototype of the corresponding function is therefore:

```
char const *String::somefun() const;
```

a `const` parameterless function in the class `String`, returning a `char constt *`.

Actually, the normal procedure for constructing pointers can still be applied:

- put parentheses around the function name (now including the class name):

```
char const * ( String::somefun ) () const
```

- Put a pointer (a star `*`) character immediately before the function-name itself:

```
char const * ( String:: * somefun ) () const
```

- Replace the function name with the name of the pointer variable:

```
char const * (String::*sp)() const
```

Another example, this time to a pointer to a data member. Assume the class `String` contains a `string text` member. How to construct a pointer to this member? Again we follow the basic procedure:

- put parentheses around the variable name (now including the class name):

```
string (String::text)
```

- Put a pointer (a star (\*)) character immediately before the variable-name itself:

```
string (String::*text)
```

- Replace the variable name with the name of the pointer variable:

```
string (String::*tp)
```

In this case, the parentheses are superfluous and may be omitted:

```
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 pointer to a member function:

- Global pointer: `char const * (*sp)() const`
- Pointer to member: `char const * (String::*sp)() const`

Nothing in the above discussion forces us to define these pointers to members in the `String` class itself. The pointer to a member may be defined in the class (so it becomes a data member itself), or in another class, or as a local or global variable. In all of 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 the need for an object of the corresponding class.

Initializing or assigning an address to such a pointer does nothing but indicating to which member the pointer will point to. This addressing can be considered a *relative address*: relative to the first byte used by an object. Consequently, no object is required when pointers to members are initialized or assigned. On the other hand, while it is allowed to initialize or assign a pointer to member, it is (of course) not possible to access these members without an associated object.

In the following example initialization of and assignment to pointers to members is illustrated (for illustration purposes all members of `PointerDemo` are defined public):

```
class PointerDemo
{
public:
    unsigned get() const
    {
        return value;
    }

    unsigned
        value;
};

int main()
{
    // initialization
```

```

    unsigned (PointerDemo::*getPtr)() = &PointerDemo::get;
    unsigned PointerDemo::*valuePtr = &PointerDemo::value;

    getPtr = &PointerDemo::get;           // assignment
    valuePtr = &PointerDemo::value;
}

```

Note that actually nothing special is involved: the difference with pointers at global scope is that we're now restricting ourselves to the scope of the `PointerDemo` class. Because of this restriction all *pointer* definitions and all variables whose addresses are used must now be given the `PointerDemo` class scope.

## 15.3 Using pointers to members

In the previous section we've seen how to define pointers to member functions. In order to use these pointers, an object is *always* required. With pointers operating at global scope, the dereferencing operator `*` is used to reach the object or value the pointer points at. With pointers to objects the field selector operator operating on pointers (`->`) or the field selector operating on objects (`.`) can be used to select appropriate members.

To use a pointer to a member with an object the pointer to member field selector (`.*`) must be used. 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 used. These two operators combine the notions of, on the one hand, a field selection (the `.` and `->` parts) to reach the appropriate field in an object and, on the second hand, the notion of dereferencing: A dereference operation is used to reach the function or variable the pointer to member points at.

Using the example from the previous section, let's see how we can use the pointer to member function and the pointer to data member:

```

#include <iostream>

class PointerDemo
{
public:
    unsigned get() const
    {
        return value;
    }

    unsigned
    value;
};

int main()
{
    // initialization
    unsigned (PointerDemo::*getPtr)() = &PointerDemo::get;
    unsigned PointerDemo::*valuePtr = &PointerDemo::value;

    PointerDemo
        object,           // (1) (see text)

```

```

        *ptr = &object;

    object.*valuePtr = 12345;           // (2)
    cout << object.*valuePtr << endl;
    cout << object.value << endl;

    ptr->*valuePtr = 54321;             // (3)
    cout << object.value << endl;

    cout << (object.*getPtr)() << endl;  // (4)
    cout << (ptr->*getPtr)() << endl;
}

```

We note:

- At statement (1) a `PointerDemo` object and a pointer to such an object is defined.
- At statement (2) we specify an object, and hence the `.*` operator, to reach the member `valuePtr` points to. This member is given a value.
- At statement (3) the same member is assigned another value, but this time using the pointer to a `PointerDemo` object, and hence we use the `->*` operator.
- At statement (4) the `.*` and `->*` are once again used, but this time to call a function through a pointer to member. Realize that the function argument list has a higher priority than pointer to member field selector operator, so the latter *must* be protected by its own set of parentheses.

Pointers to members can be used profitably in situations where a class has a member which behaves differently depending on, e.g., a configuration state. Consider once again a class `Person` from section 7.2. This class contains fields for a person's name, address and phone. Let's assume we want to construct a `Person` data base of employees. The employee data base can be queried, by depending on the kind of person querying the data base 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 `getAddress()` must return something like '<not available>' in cases where the person querying the data base is not allowed to see the person's address, and the actual address in other cases.

Assume the employee data base is opened with 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 all information about the employees, the `SALESPERSON` are allowed to see the employee's phone numbers, while the `CLERK` is only allowed to see whether a person is actually a member of the organization.

We now construct a member string `getPersonInfo(char const *name)` in the data base class. A standard implementation of this class could be:

```

string PersonData::getPersonInfo(char const *name)
{
    Person
        *p = lookup(name);           // see if 'name' exists

    if (!p)
        return "not found";
}

```

```

        switch (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 be evaluated every time `getPersonCode()` is called, and similar constructions are required in other functions that might show behavior conditional to the value of the category variable.

However, we can also define a member `infoPtr` as a pointer to a member function of the class `PersonData` returning a string and expecting a `Person` pointer as its argument. Note that this pointer can now be used to point to `allInfo()`, `noPhone()` or `nameOnly()`. Furthermore, the function that the pointer variable points to will be known by the time the `PersonData` object is constructed, as the employee status is given as an argument to the constructor of the `PersonData` object.

After having set the `infoPtr` member to the appropriate member function, the `getPersonInfo()` member function may now be rewritten:

```

string PersonData::getPersonInfo(char const *name)
{
    Person
        *p = lookup(name);        // see if 'name' exists

    return p ? (this->*infoPtr)(p) : "not found";
}

```

Note the construction that is used for accessing a pointer to member within a class: `this->*infoPtr`.

The member `infoPtr` is defined as follows (within the class `PersonData`, omitting other members):

```

class PersonData
{
    private:
        string (PersonData::*infoPtr)(Person *p);
};

```

Finally, the constructor must initialize `infoPtr` to point to the current member function. The constructor could, for example, be given the following code (showing only the pertinent code):

```

PersonData::PersonData(PersonData::EmployeeCategory cat)
{
    switch (cat)
    {
        case BOARD:
        case SUPERVISOR:
            infoPtr = &PersonData::allInfo;
        case SALESPERSON:

```

```

        infoPtr = &PersonData::noPhone;
    case CLERK:
        infoPtr = &PersonData::nameOnly;
    }
}

```

Note the way the addresses of the member functions are taken: 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 given in section 17.4.60, in the context of the `stable_sort()` generic algorithm.

## 15.4 Pointers to static members

Static members of a class exist without an object of their class. In other words, they can exist *outside of* any object of their class, and they have no `this` pointer. When these static members are public, they can be accessed as a global function, albeit that their class names are required when they are called.

Assume that a class `String` has a public static member function `int n_strings()`, returning the number of string objects created so far. Then, without using any `String` object the function `String::n_strings()` may be called:

```

void fun()
{
    cout << String::n_strings() << endl;
}

```

Public static members can be treated as globally accessible functions and data. Private static members, on the other hand, can be accessed only from within the context of their class: they can only be accessed from inside the member functions of their class.

Since static members have no particular link with objects of their class, but are comparable to global functions and data, their addresses can be stored in ordinary pointer variables, operating at the global level. Actually, using a pointer to member to address a static member of a class would produce a compilation error.

For example, the address of a static member function `int String::n_strings()` can simply be stored in a variable `int (*pfi)()`, even though `int (*pfi)()` has *nothing* in common with the class `String`. This is illustrated in the next example:

```

void fun()
{
    int
        (*pfi)();

    pfi = String::n_strings;
        // address of the static member function

    cout << pfi() << endl;
        // print the value produced by
        // String::n_strings()
}

```

## Chapter 16

# 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 the surrounding class. For example, with the class `string` a type `string::iterator` is available which will provide all elements (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 (see chapter 13), if it is nested in the `private` section, it is only visible for the members of the surrounding class.

The surrounding class has no privileges with respect to the nested class. So, the nested class still has full control over the accessibility of its members by the surrounding class. For example, consider the following class definition:

```
class Surround
{
    public:
        class FirstWithin
        {
            public:
                FirstWithin();
                int getVar() const
                {
                    return (variable);
                }
            private:
                int
                    variable;
        };
    private:
        class SecondWithin
        {
            public:
                SecondWithin();
                int getVar() const
```



```

        {
            return (variable);
        }
    private:
        int
            variable;
};
};

```

In this definition access to the members is defined as follows:

- The class `FirstWithin` is visible both outside and inside `Surround`. The class `FirstWithin` has therefore global scope.
- The constructor `FirstWithin()` and the member function `getVar()` of the class `FirstWithin` are also globally visible.
- The `int variable` datamember is only visible for the members of the class `FirstWithin`. Neither the members of `Surround` nor the members of `SecondWithin` can access the variable of the class `FirstWithin` directly.
- The class `SecondWithin` is visible only 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.
- The constructor `SecondWithin()` and the member function `getVar()` of the class `SecondWithin` can also only be reached by the members of `Surround` (and by the members of its nested classes).
- The `int variable` datamember of the class `SecondWithin` is only visible for the members of the class `SecondWithin`. Neither the members of `Surround` nor the members of `FirstWithin` can access the variable of the class `SecondWithin` directly.

If the surrounding class should have access rights to the private members of its nested classes or if nested classes should have access rights to the private members of the surrounding class, the classes can be defined as friend classes (see section 16.3).

The nested classes can be considered members of the surrounding class, but the members of nested classes are *not* members of the surrounding class. So, a member of the class `Surround` may not access `FirstWithin::getVar()` directly. This is understandable considering the fact that a `Surround` object is not also a `FirstWithin` or `SecondWithin` object. The nested classes are only available as typenames. They do not imply containment as objects by the surrounding class. If a member of the surrounding class should use a (non-static) member of a nested class then a pointer to a nested class object or a nested class datamember must be defined in the surrounding class, which can thereupon be used by the members of the surrounding class to access 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()` which uses the inner object that is composed in `Outer` to call the `infunction()` member function of `Inner`:

```

class Outer
{
    public:
        void caller()
        {
            inner.infunction();
        }
};

```

```

    }
private:
    class Inner
    {
        public:
            void infunction();
    };
    Inner
        inner;
};

```

Also note that the function `Inner::infunction()` can be called as part of the inline definition of `Outer::caller()`, even though the definition of the class `Inner` is yet to be seen by the compiler.

## 16.1 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 functions that were defined outside of the class definition: if the function `Outer::caller()` would have been defined outside of the class `Outer`, the full class definition (including the definition of the class `Inner`) would have been available to the compiler. In that situation the function is perfectly compilable. Inline functions can be compiled accordingly: they can be defined and use any nested class appearing later in the class interface.

However, inline member functions can also be defined outside of their surrounding class. Consider the constructor of the class `FirstWithin` in the example of the previous section. The constructor `FirstWithin()` is defined in the class `FirstWithin`, which is, in turn, defined within the class `Surround`. Consequently, the class scopes of the two classes must be used to define the constructor. E.g.,

```

Surround::FirstWithin::FirstWithin()
{
    variable = 0;
}

```

Static (data) members can be defined accordingly. If the class `FirstWithin` would have a static unsigned datamember `epoch`, it could be initialized as follows:

```

Surround::FirstWithin::epoch = 1970;

```

Furthermore, multiple scope resolution operators are needed to refer to public static members in code outside of the surrounding class:

```

void showEpoch()
{
    cout << Surround::FirstWithin::epoch = 1970;
}

```

Inside the members of the class `Surround` only the `FirstWithin::` scope must be used; inside the members of 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
```

## 16.2 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 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. Such cross references require forward declarations:

```
class Outer
{
    private:
        class Inner2;          // forward declaration

        class Inner1
        {
            private:
                Inner2
                *pi2;          // points to Inner2 objects
        };
        class Inner2
        {
            private:
                Inner1
                *pi1;          // points to Inner1 objects
        };
};
```

## 16.3 Accessing private members in nested classes

In order to allow nested classes to access the private members of the surrounding class; to access the private members of other nested classes; or to allow the surrounding class to access the private members of nested classes, the `friend` keyword must be used. Consider the following situation, in which a class `Surround` has two nested classes `FirstWithin` and `SecondWithin`, while each class has a static data member `int` variable:

```
class Surround
{
    public:
        class FirstWithin
```

```

    {
        public:
            int getValue();
        private:
            static int
                variable;
    };
    int getValue();
private:
    class SecondWithin
    {
        public:
            int getValue();
        private:
            static int
                variable;
    };
    static int
        variable;
};

```

If the class Surround should be able to access the private members of FirstWithin and SecondWithin, these latter two classes must declare Surround to be their friend. The function Surround::getValue() can thereupon access the private members of the nested classes. For example (note the friend declarations in the two nested classes):

```

class Surround
{
    public:
        class FirstWithin
        {
            friend class Surround;
            public:
                int getValue();
            private:
                static int
                    variable;
        };
        int getValue()
        {
            FirstWithin::variable = SecondWithin::variable;
            return (variable);
        }
    private:
        class SecondWithin
        {
            friend class Surround;
            public:
                int getValue();
            private:
                static int
                    variable;
        };
        static int

```

```

        variable;
};

```

Now, in order to allow the nested classes to access the private members of the surrounding class, the class `Surround` must declare the nested classes as friends. The `friend` keyword may only be used when the class that is to become a friend is already known as a class by the compiler, so either a forward declaration of the nested classes is required, which is followed by the friend declaration, or the friend declaration follows the definition of the nested classes. The forward declaration followed by the friend declaration looks like this:

```

class Surround
{
    class FirstWithin;
    class SecondWithin;
    friend class FirstWithin;
    friend class SecondWithin;

    public:
        class FirstWithin;
};

```

Alternatively, the friend declaration may follow the definition of the classes. Note that a class can be declared a friend following its definition, while the inline code in the definition already uses the fact that it will be declared a friend of the outer class. Also note that the inline code of the nested class uses members of the surrounding class which have not yet been seen by the compiler. Finally note that `'variable'` which is defined in the class `Surround` is accessed in the nested classes as `Surround::variable`:

```

class Surround
{
    public:
        class FirstWithin
        {
            friend class Surround;
            public:
                int getValue()
                {
                    Surround::variable = 4;
                    return (variable);
                }
            private:
                static int
                    variable;
        };
        friend class FirstWithin;

        int getValue()
        {
            FirstWithin::variable = SecondWithin::variable;
            return (variable);
        }
    private:
        class SecondWithin

```

```

{
    friend class Surround;
public:
    int getValue()
    {
        Surround::variable = 40;
        return (variable);
    }
private:
    static int
        variable;
};
friend class SecondWithin;

static int
    variable;
};

```

Finally, we want to allow the nested classes to access each other's private members. Again this requires some friend declarations. In order to allow `FirstWithin` to access `SecondWithin`'s private members nothing but a friend declaration in `SecondWithin` is required. However, to allow `SecondWithin` to access the private members of `FirstWithin` the friend class `SecondWithin` declaration cannot be plainly given in the class `FirstWithin`, as the definition of `SecondWithin` has not yet been given. A forward declaration of `SecondWithin` is required, and this forward declaration must be given in the class `Surround`, rather than in the class `FirstWithin`. Clearly, the forward declaration `class SecondWithin` in the class `FirstWithin` itself makes no sense, as this would refer to an external (global) class `FirstWithin`. But the attempt to provide the forward declaration of the nested class `SecondWithin` inside `FirstWithin` as `class Surround::SecondWithin` also fails miserably, with the compiler issuing a message like

‘Surround’ does not have a nested type named ‘SecondWithin’

The proper procedure here is to declare the class `SecondWithin` in the class `Surround`, before the class `FirstWithin` is defined. Using this procedure, the friend declaration of `SecondWithin` is accepted inside the definition of `FirstWithin`. The following class definition allows full access of the private members of all classes by all other classes:

```

class Surround
{
    class SecondWithin;
public:
    class FirstWithin
    {
        friend class Surround;
        friend class SecondWithin;
public:
        int getValue()
        {
            Surround::variable = SecondWithin::variable;
            return (variable);
        }
private:
        static int

```

```

        variable;
};
friend class FirstWithin;

int getValue()
{
    FirstWithin::variable = SecondWithin::variable;
    return (variable);
}
private:
    class SecondWithin
    {
        friend class Surround;
        friend class FirstWithin;
    public:
        int getValue()
        {
            Surround::variable = FirstWithin::variable;
            return (variable);
        }
    private:
        static int
            variable;
};
friend class SecondWithin;

static int
    variable;
};

```

## 16.4 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. In the class `ios` we've seen values like `ios::beg` and `ios::cur`. These values are (i.e., in the current GNU C++ implementation) defined as values in the `seek_dir` enumeration:

```

class ios : public _ios_fields
{
    public:
        enum seek_dir
        {
            beg,
            cur,
            end
        };
};

```

For illustration purposes, let's assume that, a class `DataStructure` 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 either of the two enumeration values. The class can be defined as follows:

```

class DataStructure
{
    public:
        enum Traversal
        {
            forward,
            backward
        };
        setTraversal(Traversal mode);
    private:
        Traversal
            mode;
};

```

Within the class `DataStructure` the values of the `Traversal` enumeration can be used directly. For example:

```

void DataStructure::setTraversal(Traversal modeArg)
{
    mode = modeArg;
    switch (mode)
    {
        forward:
            break;

        backward:
            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 enough. 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:

```

void fun()
{
    DataStructure::Traversal          // enum typename required
        localMode = DataStructure::forward; // enum typename not required

    DataStructure
        ds;

    ds.setTraversal(DataStructure::backward); // enum typename not required
}

```

Again, if `DataStructure` would define a nested class `Nested` in which the enumeration `Traversal` would have been defined, the two class scopes would have been required. In that case the former example would have to be coded as follows:

```

void fun()
{
    DataStructure::Nested::Traversal

```



```

        localMode = DataStructure::Nested::forward;

        DataStructure
            ds;

        ds.setTraversal(DataStructure::Nested::backward);
    }

```

### 16.4.1 Empty enumerations

Enum types usually have values. However, this is not required. In section 14.5.1 the `std::bad_cast` type was introduced. A `std::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 `std::bad_cast` could be caught as type, disregarding any value it might represent.

Actually, it is not necessary for a type to contain values. It is possible to define an *empty enum*, an enum without any values, whose name may thereupon be used as a legitimate type name in, e.g. a catch clause defining an exception handler.

An empty enum is defined as follows (often, but not necessarily within a class):

```

enum EmptyEnum
{};

```

Now an `EmptyEnum` may be thrown (and caught) as an exception:

```

#include <iostream>

enum EmptyEnum
{};

int main()
{
    try
    {
        throw EmptyEnum();
    }
    catch (EmptyEnum)
    {
        cout << "Caught empty enum\n";
    }
}
/*
    Generated output:
    Caught empty enum
*/

```

## Chapter 17

# The Standard Template Library, generic algorithms

The Standard Template Library (STL) consists of containers, generic algorithms, iterators, function objects, allocators and adaptors. The STL is a general purpose library consisting of algorithms and data structures. The data structures that are used in the algorithms are *abstract* in the sense that the algorithms can be used on (practically) every data type.

The algorithms can work on these abstract data types due to the fact that they are *template* based algorithms. In this chapter the *construction* of these templates is not further discussed (see chapter 18 for that). Rather, the *use* of these template algorithms is the focus of this chapter.

Several parts of the standard template library have already been discussed in the C++ Annotations. In chapter 12 the abstract containers were discussed, and in section 9.9 function objects were introduced. Also, *iterators* were mentioned at several places in this document.

The remaining components of the STL will be covered in this chapter. Iterators, adaptors and generic algorithms will be discussed in the coming sections. *Allocators* take care of the memory allocation within the STL. The default allocator class suffices for most applications, and are not further discussed in the C++ Annotations.

Forgetting to delete allocated memory is a common source of errors or memory leaks in a program. The `auto_ptr` template class may be used to prevent these types of problems. The `auto_ptr` class is discussed in section 17.3 in this chapter.

### 17.1 Predefined function objects

Function objects play important roles in combination with generic algorithms. For example, there exists a generic algorithm `sort()` that takes two iterators defining the range of objects that should be sorted, and 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 `stringVec` is as simple as:

```
sort(stringVec.begin(), stringVec.end(), greater<string>());
```

The last argument is recognized as a *constructor*: it is an *instantiation* of the `greater<>()` template class, applied to `strings`. This object is called as a function object by the `sort()` generic algorithm.

The `operator()()` (function call operator) itself is *not* visible at this point: don't confuse the parentheses in `greater<string>()` with the calling of its `operator()()`. When that operator is actually used by `sort()`, it receives two arguments: two strings to compare for 'greatness'. Internally, the `operator>()` of the underlying datatype (i.e., `string`) is called by `greater<string>::operator()()` to compare the two objects. Since the function call operator of `greater<>` is defined inline, the call itself is not actually present in the code. Rather, `sort()` calls `string::operator>()`, thinking it called `greater<>::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 `string::operator<()` (for an incremental sort) is not appropriate, as it does case sensitive comparisons. So, we provide our own `case_less` class, in which the two strings are compared case insensitively. Using the standard C function `strcasecmp()`, the following program performs the trick. It sorts its command-line arguments in ascending alphabetical order:

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>

class case_less
{
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, case_less());
    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << endl;
}
```

The default constructor of the class `case_less` is used with the final argument of `sort()`. Therefore, the only member function that must be defined with the class `case_less` is the function object `operator operator()()`. Since we know it's called with `string` arguments, we define it to expect two `string` arguments, which are used in the `strcasecmp()` function. Furthermore, the `operator()()` function is made inline, so that it does not produce overhead in the `sort()` function. The `sort()` function calls the function object with various combinations of strings, i.e., it *thinks* it does so. However, in fact it calls `strcasecmp()`, due to the inline-nature of `case_less::operator()()`.

The comparison function object is often a *predefined function object*, since these are available for most of the common operations. In the following sections the available predefined function objects are presented, together with some examples showing their use. At the end of the section about function objects *function adaptors* are introduced. In order to use predefined function objects and function adaptors, sources must

```
#include <functional>
```

Predefined function objects are used predominantly with generic algorithms. Predefined function objects exist for arithmetic, relational, and logical operations. In section 19.5 predefined function objects are developed performing bitwise operations.

### 17.1.1 Arithmetic Function Objects

The arithmetic function objects support the standard arithmetic operations: addition, subtraction, multiplication, division, modulus and negation. By using predefined arithmetic function objects, the corresponding operator of the associated data type is invoked. For example, for addition the function object `plus<Type>` is available. If we set `type` to `unsigned` then the `+` operator for unsigned values is used, if we set `type` to `string`, then the `+` operator for strings is used. For example:

```
#include <iostream>
#include <string>
#include <functional>

int main(int argc, char **argv)
{
    plus<unsigned>
        uAdd;          // function object to add unsigneds

    cout << "3 + 5 = " << uAdd(3, 5) << endl;

    plus<string>
        sAdd;          // function object to add strings

    cout << "argv[0] + argv[1] = " << sAdd(argv[0], argv[1]) << endl;
}
/*
    generated output with call: a.out going

3 + 5 = 8
argv[0] + argv[1] = a.outgoing
*/
```

Why is this useful? Note that the function object can be used for all kinds of data types, not only on the predefined datatypes, in which the particular operator has been overloaded. Assume that we want to perform an operation on a common variable on the one hand and, on the other hand, on each element of an array in turn. E.g., we want to compute the sum of the elements of an array; or we want to concatenate all the strings in a text-array. In situations like these the function objects come in handy. As noted before, the function objects are heavily used in the context of the generic algorithms, so let's take a quick look ahead at one of them.

One of the generic algorithms is called `accumulate()`. It visits all elements implied 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. For example, the following program accumulates all command line arguments, and prints the final string:

```
#include <iostream>
#include <string>
#include <functional>
#include <numeric>
```

```

int main(int argc, char **argv)
{
    string
        result =
            accumulate(argv, argv + argc, string(), plus<string>());

    cout << "All concatenated arguments: " << result << endl;
}

```

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. It could as well have been initialized to

```
string("All concatenated arguments: ")
```

in which case the `cout` statement could have been a simple

```
cout << result << endl
```

Then, the operator to apply is `plus<string>()`. Here it is important to note that a constructor is called: it is *not* `plus<string>`, but rather `plus<string>()`. The final concatenated string is returned.

Now we define our own class `Time`, in which the `operator+()` has been overloaded. Again, we can apply the predefined function object `plus`, now tailored to our newly defined datatype, to add times:

```

#include <iostream>
#include <strstream>
#include <string>
#include <vector>
#include <functional>
#include <numeric>

class Time
{
    friend ostream &operator<<(ostream &str, Time const &time)
    {
        return cout << time.days << " days, " << time.hours << ":" <<
            time.minutes << ":" << time.seconds;
    }

public:
    Time(unsigned hours, unsigned minutes, unsigned seconds)
    {
        days = 0;
        this->hours = hours;
        this->minutes = minutes;
        this->seconds = seconds;
    }

    Time(Time const &other)

```

```

    {
        this->days    = other.days;
        this->hours    = other.hours;
        this->minutes  = other.minutes;
        this->seconds  = other.seconds;
    }

    Time operator+(Time const &rValue) const
    {
        Time
            added(*this);

        added.seconds += rValue.seconds;
        added.minutes += rValue.minutes + added.seconds / 60;
        added.hours   += rValue.hours   + added.minutes / 60;
        added.days     += rValue.days     + added.hours   / 24;
        added.seconds  %= 60;
        added.minutes  %= 60;
        added.hours    %= 24;

        return added;
    }

private:
    unsigned
        days,
        hours,
        minutes,
        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>()
        ) <<
        endl;
}
/*
    produced output:
    2 days, 14:51:30
*/

```

Note that all member functions of `Time` in the above source are inline functions. This approach was followed in order to keep the example relatively small and to show explicitly that the `operator+()` function may be an inline function. On the other hand, in real life the `operator+()` function of `Time` should probably not be made inline, due to its size. Considering the previous discussion of the `plus` function object, the example is pretty straightforward. The class `Time` defines two constructors, the second one being the copy-constructor, it defines a conversion operator (`operator char const *()`) to produce a textual representation of the stored time (deploying an `ostream` object, see chapter 5), and it defines its own `operator+()`, adding two time objects.

The overloading of `operator+()` deserves some attention. In expressions like `x + y` neither `x` nor `y` are modified. The result of the addition is returned as a temporary value, which is then used in the rest of the expression. Consequently, in the `operator+()` function the `this` object and the `rValue` object must not be modified. Hence the `const` modifier for `operator+()`, forcing `this` to be constant, and the `const` modifier for `rValue`, forcing `rValue` to be constant. The sum of both times is stored in a separate `Time` object, a copy of which is then returned by the function.

In the `main()` function four `Time` objects are stored in a `vector<Time>` object. Then, the `accumulate()` generic algorithm is called to compute the accumulated time. It returns a `Time` object, which is inserted in the `cout ostream` object.

While the first example did show the use of a *named* function object, the last two examples showed the use of *anonymous* objects which were passed to the `(accumulate())` function.

The following arithmetic objects are available as predefined objects:

- `plus<>()`, as shown, this object calls the `operator+()`
- `minus<>()`, calling `operator-()` as a binary operator,
- `multiplies<>()`, calling `operator*()` as a binary operator,
- `divides<>()`, calling `operator/()`,
- `modulus<>()`, calling `operator%()`,
- `negate<>()`, calling `operator-()` as a unary operator.

An example using the unary `operator-()` follows, in which the `transform()` generic algorithm is used to toggle the signs of all elements in an array. The `transform()` generic algorithm 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.

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>

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 << endl;
}
/*
    generated output:
    -1, 2, -3, 4, -5, 6,
*/

```

### 17.1.2 Relational Function Objects

The relational operators may be called from the relational function objects. All standard relational operators are supported: `==`, `!=`, `>`, `>=`, `<` and `<=`. The following objects are available:

- `equal_to<>()`, calling operator `==()`,
- `not_equal_to<>()`, calling operator `!=()`,
- `greater<>()`, calling operator `>()`,
- `greater_equal<>()`, calling operator `>=()`,
- `less<>()`, calling operator `<()`,
- `less_equal<>()`, calling operator `<=()`.

Like the arithmetic function objects, these function objects can be used as *named* or as *anonymous* objects. An example using the relational function objects using the generic algorithm `sort()` is:

```

#include <iostream>
#include <string>
#include <functional>
#include <algorithm>

int main(int argc, char **argv)
{
    sort(argv, argv + argc, greater_equal<string>());

    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << endl;

    sort(argv, argv + argc, less<string>());

    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << endl;
}

```

The `sort()` generic algorithm expects an iterator range and a comparator for the underlying data type. The example shows the alphabetic sorting of strings and the reversed sorting of strings. 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 the type of the elements of `argv` is `char *`, and that the relational function object expects a string. The relational object `greater_equal<string>()` will therefore use the `>=` operator of strings, but will be called with `char *` variables. The conversion from `char *` arguments to `string` `const &` parameters is done implicitly by the `string(char const *)` constructor.

### 17.1.3 Logical Function Objects

The logical operators are called by the logical function objects. The standard logical operators are supported: `&&`, `||` and `!`. The following objects are available:

- `logical_and<>()`, calling operator `&&()`,
- `logical_or<>()`, calling operator `||()`,
- `logical_not<>()`, calling operator `!()` (unary operator).

An example using the operator `!()` is the following trivial example, in which the `transform()` generic algorithm is used to transform the logical values stored in an array:

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>

int main(int argc, char **argv)
{
    bool
        bArr[] = {true, true, true, false, false, false};
    unsigned const
        bArrSize = sizeof(bArr) / sizeof(bool);

    for (int idx = 0; idx < bArrSize; ++idx)
        cout << bArr[idx] << " ";
    cout << endl;

    transform(bArr, bArr + bArrSize, bArr, logical_not<bool>());

    for (int idx = 0; idx < bArrSize; ++idx)
        cout << bArr[idx] << " ";
    cout << endl;
}
/*
    generated output:
    1 1 1 0 0 0
    0 0 0 1 1 1
*/
```

### 17.1.4 Function Adaptors

Function adaptors modify the working of existing function objects. There are two kinds of function adaptors:

- *Binders* are function adaptors converting binary function objects to unary function objects. They do so by *binding* one object to a constant function object. For example, with the `minus<int>()` function object, which is a binary function object, the first argument may be bound to 100, meaning that the resulting value will always be 100 minus the value of the second argument. Either the first or the second argument may be bound to a specific value. To bind the first argument to a specific value, the function object `bind1st()` is used. To bind the second argument of a binary function to a specific value `bind2nd()` is used. As an example, assume we want to count all elements of a vector of `Person` objects that exceed (according to some criterion) some reference `Person` object. For this situation we pass the following binder and relational function object to the `count_if()` generic algorithm:

```
bind2nd(greater<Person>(), referencePerson)
```

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 given to the predicate, which is therefore a unary function. By using the binder the binary function object `greater()` is adapted to a unary function object, comparing each of the elements in the range to the reference person. Here is, to be complete, the call of the `count_if()` function:

```
count_if(pVector.begin(), pVector.end(),
         bind2nd(greater<Person>(), referencePerson))
```

- The *negators* are function adaptors converting the truth value of a predicate function. Since there are unary and binary predicate functions, there are two negator function adaptors: `not1()` is the negator to be used with unary function objects, `not2()` is the negator to be used with binary function objects.

If we want to count the number of persons in a `vector<Person>` vector *not* exceeding a certain reference person, we may, among other approaches, use either of the following alternatives:

- Use a binary predicate that directly offers the required comparison:

```
count_if(pVector.begin(), pVector.end(),
         bind2nd(less_equal<Person>(), referencePerson))
```

- Use `not2` in combination with the `greater()` predicate:

```
count_if(pVector.begin(), pVector.end(),
         bind2nd(not2(greater<Person>()), referencePerson))
```

- Use `not1()` in combination with the `bind2nd()` predicate:

```
count_if(pVector.begin(), pVector.end(),
         not1(bind2nd(greater<Person>(), referencePerson)))
```

The following little example illustrates the use of negator function adaptors, completing the section on function objects:

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
```

```

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)) <<
        endl;
    cout << count_if(iArr, iArr + 10, bind2nd(not2(greater<int>()), 6)) <<
        endl;
    cout << count_if(iArr, iArr + 10, not1(bind2nd(greater<int>(), 6))) <<
        endl;
}
/*
    produced output:
    6
    6
    6
    */

```

## 17.2 Iterators

Iterators are an abstraction of pointers. In general, the following holds true for iterators:

- Given an iterator `iter`, `*iter` represents the object the iterator points to (alternatively, `iter->` can be used to reach 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 have a *reversed\_iterator* type, in which the `iter++` operation actually reaches an previous element in a sequence.
- For the containers that have their elements stored consecutively in memory *pointer arithmetic* is available as well. This includes the `vector` and `deque`. For these containers `iter + 2` points to the second element beyond the one to which `iter` points.

The STL containers usually define members producing iterators (i.e., type `iterator`) using member functions `begin()` and `end()` and, in the case of reversed iterators (type `reverse_iterator`), `rbegin()` and `rend()`. Standard practice requires the iterator range to be *left inclusive*: the notation `[left, right)` indicates that `left` is an iterator pointing to the first element that is to be considered, while `right` is an iterator pointing just *beyond* the last element to be used. The iterator-range is said to be *empty* when `left == right`.

The following example shows a situation where all elements of a vector of strings are written to `cout` using the iterator range `[begin(), end())`, and the iterator range `[rbegin(), rend())`. Note that the `for`-loops for both ranges are identical:

```

#include <iostream>
#include <vector>
#include <string>

int main(int argc, char **argv)
{
    vector<string>

```

```

        args(argv, argv + argc);

    for
    (
        vector<string>::iterator iter = args.begin();
        iter != args.end();
        ++iter
    )
        cout << *iter << " ";

    cout << endl;

    for
    (
        vector<string>::reverse_iterator iter = args.rbegin();
        iter != args.rend();
        ++iter
    )
        cout << *iter << " ";

    cout << endl;

    return (0);
}

```

Furthermore, the STL defines *const\_iterator* types to be able to visit a range of the 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\_iterator*s are required:

```

#include <iostream>
#include <vector>
#include <string>

int main(int argc, char **argv)
{
    const vector<string>
        args(argv, argv + argc);

    for
    (
        vector<string>::const_iterator iter = args.begin();
        iter != args.end();
        ++iter
    )
        cout << *iter << " ";

    cout << endl;

    for
    (
        vector<string>::const_reverse_iterator iter = args.rbegin();
        iter != args.rend();
        ++iter
    )

```

```

    )
    cout << *iter << " ";

    cout << endl;
}

```

The examples also illustrates the use of plain pointers for 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 `sarg` with, `argv + argc` points just beyond the last element to be used, `argv++` reaches the next string. This is a general characteristic of pointers, which is why they too can be used in situations where iterators are expected.

The STL defines five types of iterators. These types recur in the generic algorithms, and in order to be able to create a particular type of iterator yourself it is important to know their characteristic. In general, iterators must define:

- `operator==()`, testing two iterators for equality,
- `operator++()`, incrementing the iterator, as prefix operator,
- `operator*()`, to access the element the iterator refers to,

The following types of iterators are used with the generic algorithms:

- **InputIterators:** InputIterators can read from a container. The dereference operator is guaranteed to work as `rvalue` in expressions. Instead of an InputIterator it is also possible to (see below) use a Forward-, Bidirectional- or RandomAccessIterator. With the generic algorithms presented in this chapter, iterator types like `InputIterator1` and `InputIterator2` may be observed as well. In these cases numbers are used to indicate which iterators ‘belong together’. E.g., the generic function `inner_product()` has the following prototype:

```

Type inner_product(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, Type init);

```

Here `InputIterator1 first1` and `InputIterator1 last1` are a set of input iterators defining one range, while `InputIterator2 first2` defines the beginning of a second range. Notations like this may be observed with other iterator types as well.

- **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, a Forward-, Bidirectional- or RandomAccessIterator.
- **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 a Bidirectional- or RandomAccessIterator.
- **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 a RandomAccessIterator. For example, to traverse a list or a deque a BidirectionalIterator may be useful.
- **RandomAccessIterators:** RandomAccessIterators provide random access to container elements. An algorithm such as `sort()` requires a RandomAccessIterator, and can therefore *not* be used with lists or maps, which only provide BidirectionalIterators.

The example given with the `RandomAccessIterator` provides an approach towards iterators: look for the iterator that's required by the (generic) algorithm, and then see whether the datastructure supports the required iterator. If not, the algorithm cannot be used with the particular datastructure.

### 17.2.1 Insert iterators

Generic algorithms often require a target container into which the results of the algorithm are deposited. For example, the `copy()` algorithm has three parameters, the first two of them define the range of elements which are visited, and the third parameter defines the first position where the result of the copy operation is to be stored. With the `copy()` algorithm the number of elements that are copied are normally available beforehand, since the number is normally equal to the number of elements in the range defined by the first two parameters, but this does not always hold true. Sometimes the number of resulting elements is different from the number of elements in the initial range. The generic algorithm `unique_copy()` is a case in point: the number of elements which are copied to the destination container is normally not known beforehand. In situations like these, an `inserter` adaptor function may be used to create elements in the destination container when they are needed.

There are three `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`:

```
copy(source.rbegin(), source.rend(), back_inserter(destination));
```

- `front_inserter()` calls the container's `push_front()` member to add 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):

```
copy(source.begin(), source.end(), front_inserter(destination));
```

- `inserter()` calls the container's `insert()` member to add 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`.

```
copy(source.begin(), source.end(), inserter(destination,
destination.begin()));
```

### 17.2.2 istream iterators

The `istream_iterator<Type>()` can be used to define an iterator (pair) for an `istream` object. The general form of the `istream_iterator<Type>()` iterator is:

```
istream_iterator<Type>
    identifier(istream &inStream)
```

Here, `Type` is the type of the data elements that are to be read from the `istream` stream. `Type` may be any type for which `operator>>()` is defined with `istream` objects.

The default constructor defines the end of the iterator pair, corresponding to end-of-stream. For example,

```
istream_iterator<string>
    endOfStream;
```

Note that the actual *stream* object which is specified for the begin-iterator is *not* mentioned.

Using a `back_inserter()` and a set of `istream_iterator<>()` adaptors all strings could be read from `cin` as follows:

```
#include <algorithm>
#include <iterator>
#include <string>
#include <vector>

int main()
{
    vector<string>
        vs;

    copy(istream_iterator<string>(cin), istream_iterator<string>(),
        back_inserter(vs));

    for
    (
        vector<string>::iterator from = vs.begin();
        from != vs.end();
        ++from
    )
        cout << *from << " ";

    cout << endl;
}
```

In the above example, note the use of the anonymous versions of the `istream_iterator` adaptors. Especially note the use of the anonymous default constructor. Instead of using `istream_iterator<string>()` the following (non-anonymous) construction could have been used:

```
istream_iterator<string>
    eos;

copy(istream_iterator<string>(cin), eos, back_inserter(vs));
```

In order to use the `istream_iterator` adaptor, sources must

```
#include <iterator>
```

This is automatically done when `iostream` is included.

## istreambuf iterators

For `streambuf` objects iterators are also available. The `istreambuf_iterator` is available after using the preprocessor directive

```
#include <iterator>
```

The `istreambuf_iterator` is available for reading from `streambuf` objects that are available for input operations. The standard operations that are also available for `istream_iterator` objects are also available for `istreambuf_iterators`. There are three constructors:

- `istreambuf_iterator<Type>()`:

This constructor represents the end-of-stream iterator while extracting values of type `Type` from the `streambuf`.

- `istreambuf_iterator<Type>(istream)`:

This constructor constructs an `istream_iterator` accessing the `streambuf` of the `istream` object, used as argument of the constructor.

- `istreambuf_iterator<Type>(streambuf *)`:

This constructor constructs an `istream_iterator` accessing the `streambuf` whose address is used as argument of the constructor.

In section 17.2.3 an example is given using both `istreambuf_iterators` and `ostreambuf_iterators`.

### 17.2.3 ostream iterators

The `ostream_iterator<Type>()` can be used to define a destination iterator for an `ostream` object. The general forms of the `ostream_iterator<Type>()` iterator are:

```
ostream_iterator<Type>  
    identifier(ostream &outStream), // and:  
    identifier(ostream &outStream, char const *delimiter);
```

`Type` is the type of the data elements that are to be written to the `ostream` stream. `Type` may be any type for which `operator<<()` is defined with `ostream` objects. The latter form of the `ostream_iterators` separates the individual `Type` data elements by `delimiter` strings. The former definition does not use any delimiters.

The next example shows the use of a `istream_iterators` and an `ostream_iterator` to copy information of a file to another file. A subtlety is the statement `in.unsetf(ios::skipws)`: it resets the `ios::skipws` flag. The consequence of this is that the default behavior of `operator>>()`, to skip whitespace, is modified. White space characters are simply returned by the operator, and the file is copied unrestrictedly. Here is the program:

```
#include <algorithm>  
#include <fstream>  
#include <iomanip>
```



```

int main(int argc, char **argv)
{
    ifstream
        in(argv[1]);

    in.unsetf(ios::skipws);

    ofstream
        out(argv[2]);

    copy(istream_iterator<char>(in), istream_iterator<char>(),
        ostream_iterator<char>(out));
}

```

In order to use the `ostream_iterator` adaptor, sources must

```
#include <iterator>
```

This is automatically done when `iostream` is included.

### **ostreambuf iterators**

The `ostreambuf_iterator` is available after using the preprocessor directive

```
#include <iterator>
```

The `ostreambuf_iterator` is available for writing to `streambuf` objects that support output operations. The standard operations that are also available for `ostream_iterator` objects are also available for `ostreambuf_iterator`s. There are two constructors:

- `ostreambuf_iterator<Type>(istream):`  
This constructor constructs an `ostream_iterator` accessing the `streambuf` of the `ostream` object, used as argument of the constructor, to insert values of type `Type`.
- `ostreambuf_iterator<Type>(streambuf *):`  
This constructor constructs an `ostream_iterator` accessing the `streambuf` whose address is used as argument of the constructor.

Here is an example using both `istreambuf_iterator`s and an `ostreambuf_iterator`, showing yet another way to copy a stream:

```

#include <iostream>
#include <algorithm>    // implies #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);
    }

```

## 17.3 The 'auto\_ptr' class

One of the problems using pointers is that strict bookkeeping is required about the memory the pointers point to. When a pointer variable goes out of scope, the memory pointed to by the pointer is suddenly inaccessible, and the program suffers from a memory leak. For example, in the following function `fun()`, a memory leak is created by calling `fun()`: 200 `int` values remain inaccessibly allocated:

```

void fun()
{
    new int[200];
}

```

The standard way to prevent memory leakage is strict bookkeeping: the programmer has to make sure that the memory pointed to by a pointer is deleted just before the pointer variable goes out of scope. In the above example the repair would be:

```

void fun()
{
    delete new int[200];
}

```

Now `fun()` only wastes a bit of time.

When a pointer variable is used to point to a *single value or object*, the bookkeeping becomes less of a burden when the pointer variable is defined as an `auto_ptr` object. In order to use the `auto_ptr` template class, sources must

```

#include <memory>

```

Normally, an `auto_ptr` object is initialized to point to a dynamically created value or object. When the `auto_ptr` object goes out of scope, the memory pointed to by the object is automatically deleted, taking over the programmer's responsibility to delete memory.

Note that:

- the `auto_ptr` object cannot be used to point to arrays of objects.
- an `auto_ptr` object should only point to memory that was made available dynamically, as only dynamically allocated memory can be deleted.
- multiple `auto_ptr` objects should not be allowed to point to the same block of dynamically allocated memory. Once an `auto_ptr` object goes out of scope, it deletes the memory it points to, immediately changing the other objects into wild pointers. Ways to prevent this situation are discussed below.

The class `auto_ptr` defines several member functions which can be used to access the pointer itself or to have the `auto_ptr` point to another block of memory. These member functions and ways to construct `auto_ptr` objects are discussed in the following sections.

### 17.3.1 Defining `auto_ptr` variables

There are three ways to define `auto_ptr` objects. Each definition contains the usual `<type>` specifier between pointed brackets. Concrete examples are given in the coming sections, but an overview of the various possibilities is presented here:

- The basic form initializes an `auto_ptr` object to a block of memory that's allocated by the `new` operator:

```
auto_ptr<type>
    identifier (new-expression);
```

This form is discussed in section 17.3.2.

- Another form initializes an `auto_ptr` object through another `auto_ptr` object:

```
auto_ptr<type>
    identifier(another auto_ptr for type);
```

This form is discussed in section 17.3.3.

- The third form simply creates an `auto_ptr` object that does not point to a particular block of memory:

```
auto_ptr<type>
    identifier;
```

This form is discussed in section 17.3.4.

### 17.3.2 Pointing to a newly allocated object

The basic form to initialize an `auto_ptr` object is to provide its constructor with a block of memory that's allocated by operator `new` operator. The generic form is:

```
auto_ptr<type>
    identifier(new-expression);
```

For example, to initialize an `auto_ptr` to a `string` variable the following construction can be used:

```
auto_ptr<string>
    strPtr(new string("Hello world"));
```

To initialize an `auto_ptr` to a `double` variable the next construction can be used:

```
auto_ptr<double>
    dPtr(new double(123.456));
```

Note the use of operator `new` in the above expressions. The use of `new` ensures the dynamic nature of the memory pointed to by the `auto_ptr` objects and allows the deletion of the memory once `auto_ptr` objects go out of scope. Also note that the type does *not* contain the pointer: the type used in the `auto_ptr` construction is the same type as used in the `new` expression.

In the example allocating 200 `int` values given in section 17.3, the memory leak can be avoided by using `auto_ptr` objects:

```
#include <memory>

void fun()
{
    auto_ptr<int>
        ip(new int[200]);
}
```

All member functions that are available for objects that were allocated by the `new` expression can be reached via the `auto_ptr` as if it was a plain pointer to the dynamically allocated object. For example, in the following program the text `'C++'` is inserted behind the word `'hello'`:

```
#include <iostream>
#include <string>
#include <memory>

int main()
{
    auto_ptr<string>
        sp(new string("Hello world"));

    cout << *sp << endl;

    sp->insert(strlen("Hello "), "C++ ");
    cout << *sp << endl;
}
/*
    produced output:
    Hello world
    Hello C++ world
*/
```

### 17.3.3 Pointing to another `auto_ptr`

Another form to initialize an `auto_ptr` object is to initialize it from another `auto_ptr` object for the same type. The generic form is:

```
auto_ptr<type>
    identifier(other auto_ptr object);
```

For example, to initialize an `auto_ptr` to a `string` variable, given the `sp` variable defined in the previous section, the following construction can be used:

```
auto_ptr<string>
```

```
    strPtr(sp);
```

A comparable construction can be used with the assignment operator in expressions. One `auto_ptr` object may be assigned to another `auto_ptr` object of the same type. For example:

```
#include <iostream>
#include <memory>
#include <string>

int main()
{
    auto_ptr<string>
        hello1(new string("Hello world")),
        hello2(hello1),
        hello3,

    hello3 = hello2;
    cout << *hello1 << endl <<
         *hello2 << endl <<
         *hello3 << endl;
}
/*
    produced output:
Segmentation fault
*/
```

Looking at the above example, we see that `hello1` is initialized as described in the previous section. Next `hello2` is defined, and it receives its value from `hello1`, using a copy constructor type of initialization. Then `hello3` is defined as a default `auto_ptr<string>`, but it receives its value through an assignment from `hello2`.

Unfortunately, the program generates a *segmentation fault*. The reason for this is that eventually *only* `hello3` points at the string `hello world`. First, at the initialization of `hello2`, `hello1` loses the memory it points. Second, at the assignment to `hello3` by `hello2`, `hello2` loses the memory it points to: once a `auto_ptr` object is used for the initialization of or assignment to another `auto_ptr` object, the rvalue `auto_ptr` object will not refer anymore to the allocated memory. The allocated memory is now 'owned' by the lvalue object.

### 17.3.4 Creating a plain `auto_ptr`

We've already seen the third form to create an `auto_ptr` object: Without arguments an empty `auto_ptr` object is constructed that does not point to a particular block of memory:

```
auto_ptr<type>
    identifier;
```

In this case the underlying pointer is set to 0 (zero). Since the `auto_ptr` object itself is not the pointer, its value cannot be compared to 0 to see if it has not been initialized. E.g., code like

```
auto_ptr<int>
    ip;
```

```

if (!ip)
    cout << "0-pointer with an auto_ptr object ?" << endl;

```

will not produce any output (actually, it won't compile either...). So, how do we inspect the value of the pointer that's maintained by the `auto_ptr` object? For this the member `get()` is available. This member function, as well as the other member functions of the class `auto_ptr` are described in the following sections.

### 17.3.5 Auto\_ptr: operators and members

The following operators are defined for the class `auto_ptr`:

- `auto_ptr &auto_ptr<Type>operator=(auto_ptr<Type> &other):`  
This operator will transfer the memory pointed to by the rvalue `auto_ptr` object to the lvalue `auto_ptr` object. So, the rvalue object *loses* the memory it pointed at.
- `Type &auto_ptr<Type>operator*():`  
This operator returns a reference to the information stored in the `auto_ptr` object. It acts like a normal dereference operator with pointers.
- `Type *auto_ptr<Type>operator->():`  
This operator returns a pointer to the information stored in the `auto_ptr` object. Through this operator members of a stored object can be selected. For example:  

```

auto_ptr<string>
sp(new string("hello"));

cout << sp->c_str() << endl;

```

The following member functions are defined for `auto_ptr` objects:

- `Type *auto_ptr<Type>::get():`  
This operator does the same as `operator->()`: it returns a pointer to the information stored in the `auto_ptr` object. This pointer can be inspected: if it's zero the `auto_ptr` object does not point to any memory. This member cannot be used to let the `auto_ptr` object point to (another) block of memory.
- `Type *auto_ptr<Type>::release():`  
This operator returns a pointer to the information stored in the `auto_ptr` object, which loses the memory it pointed at. The member can be used to transfer the information stored in the `auto_ptr` object to a plain `Type` pointer. After using `release()`, the `auto_ptr` object contains a 0-pointer while it is the responsibility of the programmer to delete the memory returned by this member function.
- `void auto_ptr<Type>::reset(Type *):`  
This operator can be called *without* argument, to delete the memory stored in the `auto_ptr` object, or with an pointer to a dynamically allocated block of memory, which will thereupon become the memory that is stored in the `auto_ptr` object. This member function can therefore be used to assign a new block of memory (new content) to an `auto_ptr` object.

## 17.4 The Generic Algorithms

The following sections describe the generic algorithms in alphabetical order. For each algorithm the following information is provided:

- The required header s
- The function prototype
- A short description
- A short example

In the prototypes of the algorithms `Type` is used to specify a generic data type. The particular type of iterator (see section 17.2) that is required is mentioned, and possibly other generic types, e.g., performing `BinaryOperations`, like `plus<Type>()`.

Almost every generic algorithm has as its first two arguments an iterator range `[first, last)`, defining the range of elements on which the algorithm operates. The iterators point to objects or values. When an iterator points to a `Type` value or object, called function objects receive `Type const &` objects or values: function objects can therefore not modify the objects they receive as their arguments.

Generic algorithms may be categorized. In the C++ Annotations the following categories of generic algorithms are distinguished:

- Comparators: comparing (ranges of) elements:

Requires: `#include <algorithm>`  
`equal(); includes(); lexicographical_compare(); max(); min(); mismatch();`

- Copiers: performing copy operations:

Requires: `#include <algorithm>`  
`copy(); copy_backward(); partial_sort_copy(); remove_copy(); remove_copy_if(); replace_copy(); replace_copy_if(); unique_copy();`

- Counters: performing count operations:

Requires: `#include <algorithm>`  
`count(); count_if();`

- Heap operators: manipulating a max-heap:

Requires: `#include <algorithm>`  
`make_heap(); pop_heap(); push_heap(); sort_heap();`

- Initializers: initializing data:

Requires: `#include <algorithm>`  
`fill(); fill_n(); generate(); generate_n();`

- Operators: performing arithmetic operations of some sort:

Requires: `#include <numeric>`  
`adjacent_difference(); accumulate(); inner_product(); partial_sum();`

- Searchers: performing search (and find) operations:

Requires: `#include <algorithm>`  
`adjacent_find(); binary_search(); equal_range(); find(); find_if(); find_end(); find_first_of();`  
`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):

Requires: `#include <algorithm>`  
`inplace_merge(); iter_swap(); merge(); next_permutation(); nth_element(); partial_sort();`  
`partial_sort_copy(); partition(); prev_permutation(); random_shuffle(); remove(); re-`  
`move_copy(); remove_if(); remove_copy_if(); reverse(); reverse_copy(); rotate(); ro-`  
`tate_copy(); sort(); stable_partition(); stable_sort(); swap(); unique();`

- Visitors: visiting elements in a range:

Requires: `#include <algorithm>`  
`for_each(); replace(); replace_copy(); replace_if(); replace_copy_if(); transform(); unique_copy();`

### 17.4.1 `accumulate()`

- Header file:

`#include <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:

```
#include<numeric>
#include<vector>
#include<iostream>

int main()
{
    int
        ia[] = {1, 2, 3, 4};
    vector<int>
        iv(ia, ia + 4);

    cout <<
        "Sum of values: " << accumulate(iv.begin(), iv.end(), int()) <<
```



```

endl <<
"Product of values: " << accumulate(iv.begin(), iv.end(), int(1),
                                   multiplies<int>()) <<
endl;
}
/*
    generated output:
    Sum of values: 10
    Product of values: 24
*/

```

### 17.4.2 adjacent\_difference()

- Header file:

```
#include <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 `op` applied to the corresponding element in the input range (left operand) and its previous element (right operand).

- Example:

```

#include<numeric>
#include<vector>
#include<iostream>

int main()
{
    int
        ia[] = {1, 2, 5, 10};
    vector<int>
        iv(ia, ia + 4),
        ov(iv.size());

    adjacent_difference(iv.begin(), iv.end(), ov.begin());

    copy(ov.begin(), ov.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}

```

```

    adjacent_difference(iv.begin(), iv.end(), ov.begin(), minus<int>());

    copy(ov.begin(), ov.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}
/*
    generated output:
    1 1 3 5
    1 1 3 5
*/

```

### 17.4.3 adjacent\_find()

- Header file:

```
#include <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, 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:

```

#include<algorithm>
#include<string>
#include<iostream>

class SquaresDiff
{
public:
    SquaresDiff(unsigned minimum): minimum(minimum)
    {}
    bool operator()(unsigned first, unsigned second)
    {
        return (second * second - first * first >= minimum);
    }
private:
    unsigned
        minimum;
};

int main()
{

```

```

string
    sarr[] =
    {
        "Alpha", "bravo", "charley", "delta", "echo", "echo",
        "foxtrot", "golf"
    };
string
    *last = sarr + sizeof(sarr) / sizeof(string),
    *result = adjacent_find(sarr, last);

cout << *result << endl;
result = adjacent_find(++result, last);

cout << "Second time, starting from the next position:\n" <<
    (
        result == last ?
            "*** No more adjacent equal elements ***"
        :
            "*result"
    ) << endl;

unsigned
    *ires,
    iv[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
    *ilast = iv + sizeof(iv) / sizeof(unsigned);

ires = adjacent_find(iv, ilast, SquaresDiff(10));
cout <<
    "The first numbers for which the squares differ at least 10: "
    << *ires << " and " << *(ires + 1) << endl;
}
/*
    generated output:
echo
Second time, starting from the next position:
** No more adjacent equal elements **
The first numbers for which the squares differ at least 10: 5 and 6
*/

```

#### 17.4.4 `binary_search()`

- Header file:

```
#include <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 range 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 range 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:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <functional>

int main()
{
    string
        sarr[] =
        {
            "alpha", "bravo", "charley", "echo", "delta",
            "foxtrot", "golf", "hotel"
        };
    string
        *last = sarr + sizeof(sarr) / sizeof(string);
    bool
        result = binary_search(sarr, last, "foxtrot");

    cout << (result ? "found " : "didn't find ") << "foxtrot" << endl;

    reverse(sarr, last);                // reverse the order of elements
                                        // binary search now fails:
    result = binary_search(sarr, last, "foxtrot");
    cout << (result ? "found " : "didn't find ") << "foxtrot" << endl;
                                        // ok when using appropriate
                                        // comparator:
    result = binary_search(sarr, last, "foxtrot", greater<string>());
    cout << (result ? "found " : "didn't find ") << "foxtrot" << endl;
}
/*
    generated output:
found foxtrot
didn't find foxtrot
found foxtrot
*/
```

### 17.4.5 `copy()`

- Header file:

```
#include <algorithm>
```

- Function prototype:

– `OutputIterator copy(InputIterator first, InputIterator last, OutputIterator destination);`

- Description:

– The range of elements implied by the iterator range `[first, last)` are 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). In the example, note the second call to `copy()`. It uses an `ostream_iterator` for string objects. This iterator will write the string values to the specified `ostream` (i.e., `cout`), separating the values by the specified separation string (i.e., `" "`).

- Example:

```
#include <algorithm>
#include <string>
#include <iostream>

int main()
{
    string
        sarr[] =
        {
            "alpha", "bravo", "charley", "echo", "delta",
            "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 << endl;
}
/*
    generated output:
    charley echo delta foxtrot golf hotel golf hotel
*/
```

- See also: `unique_copy()`

#### 17.4.6 `copy_backward()`

- Header file:

`#include <algorithm>`

- Function prototype:

– `BidirectionalIterator copy_backward(InputIterator first, InputIterator last, BidirectionalIterator last2);`

- Description:

- The range 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)`. The return value is the `BidirectionalIterator` pointing at 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>

int main()
{
    string
        sarr[] =
        {
            "alpha", "bravo", "charley", "echo", "delta",
            "foxtrot", "golf", "hotel"
        };
    string
        *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        copy_backward(sarr + 3, last, last - 3),
        last,
        ostream_iterator<string>(cout, " ")
    );
    cout << endl;
}
/*
    generated output:
    golf hotel foxtrot golf hotel foxtrot golf hotel
*/
```

## 17.4.7 count()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

- `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. To determine whether `value` is equal to an element in the iterator range `Type::operator==( )` is used.

- Example:

```
#include<algorithm>
#include<iostream>

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) <<
        endl;
}
/*
    generated output:
    Number of times the value 3 is available: 3
*/
```

#### 17.4.8 count\_if()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

```
- 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:

```
#include<algorithm>
#include<iostream>

class Odd
{
public:
    bool operator()(int value)
    {
        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()) <<
        endl;
    }
    /*
        generated output:
        The number of odd values in the array is: 5
    */

```

### 17.4.9 equal()

- Header file:

```
#include <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:

```

#include <algorithm>
#include <string>
#include <iostream>

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", "echo", "Delta",
            "foxtrot", "Golf", "hotel"

```



```

    },
    second[] =
    {
        "alpha", "bravo", "charley", "echo", "delta",
        "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") <<
        endl <<
        "compared case-insensitively, they are " <<
        (
            equal(first, last, second, CaseString()) ?
            "equal" : "not equal"
        ) <<
        endl;
}
/*
    generated output:
    The elements of 'first' and 'second' are pairwise not equal
    compared case-insensitively, they are equal
*/

```

### 17.4.10 equal\_range()

- Header files:

```
#include <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 of, e.g., the `map` (section 12.2.6) and `multimap` (section 12.2.7) containers):

- The first prototype: Starting from a sorted sequence (where the `operator<()` of the underlying data type was used to sort the elements in the provided range), a pair of iterators representing the return value of, respectively, `lower_bound()` (returning the first element that is equal to the provided reference value, see section 17.4.25) and `upper_bound()` (returning the first element beyond the provided reference value, see section 17.4.66) is returned.
- 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 representing the return value of, respectively, `lower_bound()` (section 17.4.25) and `upper_bound()` (section 17.4.66) is returned.

- Example:

```
#include <algorithm>
#include <functional>
#include <iostream>

void main()
{
    int
        range[] = {1, 3, 5, 7, 7, 9, 9, 9};
    unsigned const
        size = sizeof(range) / sizeof(int);

    pair<int *, int *>
        pi;

    pi = equal_range(range, range + size, 7);

    cout << "Lower bound for 7: ";
    copy(pi.first, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "Upper bound for 7: ";
    copy(pi.second, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    sort(range, range + size, greater<int>());

    cout << "Sorted in descending order\n";

    copy(range, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    pi = equal_range(range, range + size, 7, greater<int>());

    cout << "Lower bound for 7: ";
    copy(pi.first, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "Upper bound for 7: ";
    copy(pi.second, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;
}

/*
    generated output:
    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
*/
```

### 17.4.11 fill()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

- 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 previous values stored in the range.

- Example:

```
#include<algorithm>
#include<vector>
#include<iostream>

int main()
{
    vector<int>
        iv(8);

    fill(iv.begin(), iv.end(), 8);

    copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}
/*
    generated output:
    8 8 8 8 8 8 8 8
*/
```

### 17.4.12 fill\_n()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

- 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 previous values stored in the range.

- Example:

```
#include<algorithm>
#include<vector>
#include<iostream>
```

```

int main()
{
    vector<int>
        iv(8);

    fill_n(iv.begin() + 2, 4, 8);

    copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}
/*
    generated output:
0 0 8 8 8 8 0 0
*/

```

### 17.4.13 find()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

```
- 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:

```

#include<algorithm>
#include<string>
#include<iostream>

int main()
{
    string
        sarr[] =
        {
            "alpha", "bravo", "charley", "echo", "delta"
        };
    string
        *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        find(sarr, last, "echo"), last, ostream_iterator<string>(cout, " ")
    );
    cout << endl;
}

```

```

    if (find(sarr, last, "india") == last)
    {
        cout << "'india' was not found in the range\n";
        copy(sarr, last, ostream_iterator<string>(cout, " "));
        cout << endl;
    }
}
/*
    generated output:
echo delta
'india' was not found in the range
alpha bravo charley echo delta
*/

```

#### 17.4.14 find\_if()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

```
- 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:

```

#include<algorithm>
#include<string>
#include<iostream>

class CaseName
{
public:
    CaseName(char const *str): _string(str)
    {}
    bool operator()(string const &element)
    {
        return (!strcasecmp(element.c_str(), _string.c_str()));
    }
private:
    string
        _string;
};

int main()
{
    string
        sarr[] =

```

```

        {
            "Alpha", "Bravo", "Charley", "Echo", "Delta",
        };
string
    *last = sarr + sizeof(sarr) / sizeof(string);

copy
(
    find_if(sarr, last, CaseName("charley")),
    last, ostream_iterator<string>(cout, " ")
);
cout << endl;

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 << endl;
}
}
/*
    generated output:
Charley Echo Delta
'india' was not found in the range
Alpha Bravo Charley Echo Delta
*/

```

### 17.4.15 find\_end()

- Header file:

```
#include <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 [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:

```
#include<algorithm>
#include<string>
#include<iostream>

class Twice
{
public:
    bool operator()(unsigned first, unsigned second) const
    {
        return (first == (second << 1));
    }
};

int main()
{
    string
        sarr[] =
        {
            "alpha", "bravo", "charley", "echo", "delta",
            "foxtrot", "golf", "hotel",
            "foxtrot", "golf", "hotel",
            "india", "juliet", "kilo"
        },
        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 << endl;

    unsigned
        range[] = {2, 4, 6, 8, 10, 4, 6, 8, 10},
        nrs[]    = {2, 3, 4};

    copy
    (
        // sequence of values starting at last sequence
        // of range[] that are twice the values in nrs[]
        find_end(range, range + 9, nrs, nrs + 3, Twice()),
        range + 9, ostream_iterator<unsigned>(cout, " ")
    );
    cout << endl;
}
/*
generated output:
```

```

foxtrot golf hotel india juliet kilo
4 6 8 10
*/

```

### 17.4.16 find\_first\_of()

- Header file:

```
#include <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 [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 [first2, last2) 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, first1) is searched for the first occurrence of an element in the sequence of elements implied by [first2, last2). Each element in the range [first1, last1) is compared to each element in the range [first2, last2), and an iterator to the first element in [first1, last1) for which the binary predicate pred (receiving an the element out of the range [first1, last1) and an element from the range [first2, last2)) returns true is returned. Otherwise, last1 is returned.

- Example:

```

#include<algorithm>
#include<string>
#include<iostream>

class Twice
{
public:
    bool operator()(unsigned first, unsigned second) const
    {
        return (first == (second << 1));
    }
};

int main()
{
    string
        sarr[] =
        {
            "alpha", "bravo", "charley", "echo", "delta",

```



```

        "foxtrot", "golf", "hotel",
        "foxtrot", "golf", "hotel",
        "india", "juliet", "kilo"
    },
    search[] =
    {
        "foxtrot",
        "golf",
        "hotel"
    };
string
    *last = sarr + sizeof(sarr) / sizeof(string);

copy
(
    // sequence starting
    find_first_of(sarr, last, search, search + 3), // at 1st 'foxtrot'
    last, ostream_iterator<string>(cout, " ")
);
cout << endl;

unsigned
    range[] = {2, 4, 6, 8, 10, 4, 6, 8, 10},
    nrs[]    = {2, 3, 4};

copy          // sequence of values starting at first sequence
(
    // of range[] that are twice the values in nrs[]
    find_first_of(range, range + 9, nrs, nrs + 3, Twice()),
    range + 9, ostream_iterator<unsigned>(cout, " ")
);
cout << endl;
}
/*
    generated output:
    foxtrot golf hotel foxtrot golf hotel india juliet kilo
    4 6 8 10 4 6 8 10
*/

```

### 17.4.17 for\_each()

- Header file:

```
#include <algorithm>
```

- Function prototype:

```
– Function for_each(InputIterator first, InputIterator last, Function func);
```

- Description:

– Each of the elements implied by the iterator range [first, last) is passed in turn as a const & to the function object func. The function is not supposed to modify the elements it receives (as the used iterator is an input iterator). If the elements are to be transformed, transform() (see section 17.4.63) should be used. The 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.

- Example:

```
#include<algorithm>
#include<string>
#include<iostream>

void capitalizedOutput(string const &str)
{
    char
        *tmp = strcpy(new char[str.size() + 1], str.c_str());

        // can't use for_each here, as 'tmp' is modified
    transform(tmp + 1, tmp + str.size(), tmp + 1, tolower);

    tmp[0] = toupper(*tmp);
    cout << tmp << " ";
    delete tmp;
};

int main()
{
    string
        sarr[] =
        {
            "alpha", "BRAVO", "charley", "ECHO", "delta",
            "FOXTROT", "golf", "HOTEL",
        },
        *last = sarr + sizeof(sarr) / sizeof(string);

    for_each(sarr, last, capitalizedOutput)("that's all, folks");
    cout << endl;
}
/*
    generated output:
Alpha Bravo Charley Echo Delta Foxtrot Golf Hotel That's all, folks
*/
```

Although the function receives `const` references to objects, applications may consider it acceptable to overrule the `const &` nature of the parameter of `func.operator()`. If the parameter expects a `Type const &cref`, the function could modify the referenced object after defining:

$$\text{Type \&ref} = \text{const\_cast<Type \&>}(cref)$$

Following this definition `ref` can be used to modify the object that was passed as a reference. It is clear that we are stepping on dangerous grounds here, but if the programmer knows what he/she is doing, no damage should result from this type cast. The alternative, using the `transform()` generic algorithm involves the copying of an object, possibly into its original location. The penalty for doing so (calling a destructor and a copy constructor) might be considered more severe than using a type cast. Let it, however, be clear that we are not really *advocating* the use of a type cast with `foreach()` here. Rather, a possibility is mentioned. It is, as always, the responsibility of the programmer to use or avoid the typecast.

### 17.4.18 generate()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

```
– 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. Note that `Generator::operator()()` is not given any arguments. The example uses a well-known fact from algebra: in order to obtain the square of  $n + 1$ , add  $1 + 2 * n$  to  $n * n$ .

- Example:

```
#include<algorithm>
#include<vector>
#include<iostream>

class NaturalSquares
{
public:
    NaturalSquares(): newsqr(0), last(0)
    {}
    unsigned operator()()
    {
        // using: (a + 1)^2 == a^2 + 2*a + 1
        return newsqr += (last++ << 1) + 1;
    }

private:
    unsigned
        newsqr,
        last;
};

int main()
{
    vector<unsigned>
        uv(10);

    generate(uv.begin(), uv.end(), NaturalSquares());

    copy(uv.begin(), uv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}
/*
generated output:
1 4 9 16 25 36 49 64 81 100
*/
```

### 17.4.19 generate\_n()

- Header file:

```
#include <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:

```
#include<algorithm>
#include<vector>
#include<iostream>

class NaturalSquares
{
public:
    NaturalSquares(): newsqr(0), last(0)
    {}
    unsigned operator()()
    {
        // (a + 1)^2 == a^2 + 2*a + 1
        return (newsqr += (last++ << 1) + 1);
    }

private:
    unsigned
        newsqr,
        last;
};

int main()
{
    vector<unsigned>
        uv(10);

    generate_n(uv.begin(), 5, NaturalSquares());

    copy(uv.begin(), uv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}
/*
    generated output:
    1 4 9 16 25 0 0 0 0 0
*/
```

### 17.4.20 includes()

- Header files:

```
#include <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 be sorted, using the operator`<()` of the underlying datatype. The function returns true if every element in the second sequence `[first2, second2)` is contained in the first sequence `[first1, second1)` (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 be sorted, using the `comp` function object. The function returns true if every element in the second sequence `[first2, second2)` is contained in the first sequence `[first1, second1)` (the second range is a subset of the first range).

- Example:

```
#include <algorithm>
#include <string>
#include <iostream>

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", "echo", "delta",
            "foxtrot", "golf", "hotel"
        },
        first2[] =
        {
            "Alpha", "bravo", "Charley", "echo", "Delta",
            "foxtrot", "Golf", "hotel"
        },
        second[] =
        {
            "charley", "foxtrot", "hotel"
        };
    unsigned
        n = sizeof(first1) / sizeof(string);
```

```

cout << "The elements of 'second' are " <<
    (includes(first1, first1 + n, second, second + 3) ? "" : "not")
    << " contained in the first sequence:\n"
    "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";
}
/*
    generated output:
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
*/

```

### 17.4.21 inner\_product()

- Header files:

```
#include <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 underlying datatype.
- The second prototype: Binary operator `op2` instead of the default addition operator, and binary operator `op1` 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 final result is returned.

- Example:

```

#include <numeric>
#include <algorithm>
#include <iostream>
#include <string>

class Cat
{
public:
    Cat(string const &sep): sep(sep)
    {}
    string operator()(string const &s1, string const &s2)
    {
        return (s1 + sep + s2);
    }
private:
    string
        sep;
};

int main()
{
    unsigned
        ia1[] = {1, 2, 3, 4, 5, 6, 7},
        ia2[] = {7, 6, 5, 4, 3, 2, 1},
        init = 0;

    cout << "The sum of all squares in ";
    copy(ia1, ia1 + 7, ostream_iterator<unsigned>(cout, " "));
    cout << "is " <<
        inner_product(ia1, ia1 + 7, ia1, init) << endl;

    cout << "The sum of all cross-products in ";
    copy(ia1, ia1 + 7, ostream_iterator<unsigned>(cout, " "));
    cout << " and ";
    copy(ia2, ia2 + 7, ostream_iterator<unsigned>(cout, " "));
    cout << "is " <<
        inner_product(ia1, ia1 + 7, ia2, init) << endl;

    string
        names1[] = {"Frank", "Karel", "Piet"},
        names2[] = {"Brokken", "Kubat", "Plomp"};

    cout << "A list of all combined names in ";
    copy(names1, names1 + 3, ostream_iterator<string>(cout, " "));
    cout << "and\n";
    copy(names2, names2 + 3, ostream_iterator<string>(cout, " "));
    cout << "is:" <<
        inner_product(names1, names1 + 3, names2, string("\t"),
            Cat("\n\t"), Cat(" ")) <<
        endl;
}
/*
generated output:

```

```

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
*/

```

### 17.4.22 inplace\_merge()

- Header files:

```
#include <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 underlying data type). 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).

- Example:

```

#include <algorithm>
#include <string>
#include <iostream>

class CaseString
{
public:
    bool operator()(string const &first, string const &second) const
    {
        return strcasecmp(first.c_str(), second.c_str()) < 0;
    }
};

int main()
{
    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 << endl;

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 << endl;
}
/*
    generated output:
alpha bravo charley delta echo foxtrot golf
ALFA bravo CHARLEY DELTA ECHO foxtrot GOLF hotel
*/

```

### 17.4.23 iter\_swap()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

```
- void iter_swap(ForwardIterator1 iter1, ForwardIterator2 iter2);
```

- Description:

```
- The elements pointed to by iter1 and iter2 are swapped.
```

- Example:

```

#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    string
        first[] = {"alpha", "bravo", "charley"},
        second[] = {"echo", "foxtrot", "golf"};
    unsigned
        n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;
}

```

```

    for (unsigned idx = 0; idx < n; ++idx)
        iter_swap(first + idx, second + idx);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
Before:
alpha bravo charley
echo foxtrot golf
After:
echo foxtrot golf
alpha bravo charley
*/

```

#### 17.4.24 lexicographical\_compare()

- Header files:

```
#include <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:

– The first prototype: The corresponding pairs of elements in the ranges pointed to by `[first1, last1)` and `[first2, 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 the `operator<()` of the underlying data type),
- \* if `last1` is reached, but `last2` isn't reached yet.

False is returned in the other cases, which indicates that the first sequence is not lexicographical less than the second sequence. I.e., false is returned

- \* at the first element in the first range which is greater than the corresponding element in the second range (using the `operator<()` of the underlying data type),
- \* if `last2` is reached, but `last1` isn't reached yet.
- \* if `last1` and `last2` are reached.

– The second prototype: With this function the binary comparison operation as defined by `comp` is used instead of the underlying `operator<()`.

- Example:

```
#include <algorithm>
```

```

#include <iostream>
#include <string>

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",
        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 "
        ) <<
        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"},
        two[] = {"ALPHA", "BRAVO", "DELTA"};

    copy(one, one + 3, ostream_iterator<string>(cout, " "));
    cout << " is ordered " <<

```

```

        (
            lexicographical_compare(one, one + 3,
                                   two, two + 3, CaseString()) ?
                "before "
            :
                "beyond or at "
        );
    copy(two, two + 3, ostream_iterator<string>(cout, " "));
    cout << endl <<
        "using case-insensitive comparisons.\n";
}
/*
    generated output:
    hello is before help in the alphabet
    hello is beyond or at hello in the alphabet
    help is beyond or at hello in the alphabet
    alpha bravo charley is ordered before ALPHA BRAVO DELTA
    using case-insensitive comparisons.
*/

```

#### 17.4.25 lower\_bound()

- Header files:

```
#include <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:

- The first prototype: The sorted elements indicated by the iterator range [first, last) are searched for the first element that is not less than (i.e., greater than or equal to) value. The returned iterator marks the location in the sequence where value can be inserted without breaking the sorted order of the elements. The operator<() of the underlying datatype is used. If no such element is found, last is returned.
- The second prototype: The elements indicated by the iterator range [first, last) must have been sorted using the compfunction (-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 false is returned. If no such element is found, last is returned.

- Example:

```

#include <algorithm>
#include <iostream>
#include <functional>

int main()
{

```

```

int
    ia[] = {10, 20, 30};

cout << "Sequence: ";
copy(ia, ia + 3, ostream_iterator<int>(cout, " "));
cout << endl;

cout << "15 can be inserted before " <<
    *lower_bound(ia, ia + 3, 15) << endl;
cout << "35 can be inserted after " <<
    (lower_bound(ia, ia + 3, 35) == ia + 3 ?
    "the last element" : "???" ) << endl;

iter_swap(ia, ia + 2);

cout << "Sequence: ";
copy(ia, ia + 3, ostream_iterator<int>(cout, " "));
cout << endl;

cout << "15 can be inserted before " <<
    *lower_bound(ia, ia + 3, 15, greater<int>()) << endl;
cout << "35 can be inserted before " <<
    (lower_bound(ia, ia + 3, 35, greater<int>()) == ia ?
    "the first element " : "???" ) << endl;
}
/*
    generated output:
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
*/

```

### 17.4.26 max()

- Header file:

```
#include <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 underlying type.
- The second prototype: `one` is returned if the binary predicate `comp(one, two)` returns true, otherwise `two` is returned.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>

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";
}
/*
generated output:
Word 'second' is lexicographically last
Word 'first' is lexicographically last
Word 'SECOND' is lexicographically last
*/
```

### 17.4.27 max\_element()

- Header file:

```
#include <algorithm>
```

- 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 [first, last) is returned. The operator<() of the underlying type is used.
- 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 most often true, compared with other elements, is returned.

- Example:

```
#include <algorithm>
#include <iostream>

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) << endl;
    cout << "The max. absolute int value is " <<
        *max_element(ia, ia + 5, AbsValue()) << endl;
}
/*
    generated output:
    The max. int value is 10
    The max. absolute int value is -12
*/
```

### 17.4.28 merge()

- Header files:

```
#include <algorithm>
```

- Function prototypes:

- `OutputIterator merge(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
- `OutputIterator merge(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`

- Description:

- The first prototype: The two (sorted) ranges `[first1, last1)` and `[first2, last2)` are merged, keeping a sorted list (using the `operator<()` of the underlying data type). The final series is stored in the range starting at `result` and ending just before the `OutputIterator` that is returned by the function.
- The second prototype: The two (sorted) ranges `[first1, last1)` and `[first2, last2)` are merged, keeping a sorted list (using the boolean result of the binary comparison operator `comp`). The final series is stored in the range starting at `result` and ending just before the `OutputIterator` that is returned by the function.

- Example:

```
#include <algorithm>
#include <string>
#include <iostream>

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[] =
        {
            "alpha", "bravo", "foxtrot", "hotel", "zulu"
        },
        range2[] =
        {
            "delta", "echo", "golf", "romeo"
        },
        result[5 + 4];

    copy(result,
        merge(range1, range1 + 5, range2, range2 + 4, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string
        range3[] =
        {
            "ALPHA", "bravo", "foxtrot", "HOTEL", "ZULU"
        },
        range4[] =
        {
            "delta", "ECHO", "GOLF", "romeo"
        };

    copy(result,
        merge(range3, range3 + 5, range4, range4 + 4, result,
            CaseString()),
        ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
alpha bravo delta echo foxtrot golf hotel romeo zulu
ALPHA bravo delta ECHO foxtrot GOLF HOTEL romeo ZULU
*/
```



\*/

### 17.4.29 min()

- Header file:

```
#include <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 underlying type.
- 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>

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";

    cout << "Word '" << min(string("first"), string("SECOND"),
        CaseString()) << "' is lexicographically first\n";
}
/*
    generated output:
Word 'first' is lexicographically first
Word 'SECOND' is lexicographically first
Word 'first' is lexicographically first
*/
```

### 17.4.30 min\_element()

- Header file:

```
#include <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 [first, last) is returned. The operator<() of the underlying type is used.
- 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 with which comp returns most often false is returned.

- Example:

```
#include <algorithm>  
#include <iostream>  
  
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) <<  
        endl;  
    cout << "The minimum absolute int value is " <<  
        *min_element(ia, ia + 5, AbsValue()) << endl;  
}  
/*  
    generated output:  
The minimum int value is -12  
The minimum absolute int value is -2  
*/
```

### 17.4.31 mismatch()

- Header files:

```
#include <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 underlying data type. 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 less elements than the first sequence.
- The second prototype: The two sequences of elements starting at `first1` and `first2` are compared using With this function the binary comparison operation as defined by `comp` is used instead of the underlying `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 less elements than the first sequence.

- Example:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <utility>

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[] =
        {
            "alpha", "bravo", "foxtrot", "hotel", "zulu"
        },
        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";
}
/*
    generated output:
    The elements hotel and Hotel at offset 3 differ
    When compared case-insensitively they match
*/

```

### 17.4.32 next\_permutation()

- Header files:

```
#include <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() re-orders 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 a that each new permutation represents the next bigger value (132 is bigger than 123, 213 is bigger than 132, etc., using operator<() of the underlying data type. 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 according to the operator<() of the underlying data type.

- The second prototype: The next permutation given the sequence of elements in the range [first, last) is determined. 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 resulting sequence would haven been ordered, using the binary predicate comp to compare elements. )
- Example:

```
#include <algorithm>
```

```

#include <iostream>
#include <string>

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 << endl;
    }
    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 << endl;
    }
    while (next_permutation(saints, saints + 4, CaseString()));
}
/*
    generated output (only partially given):
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
    ...
    */

```

### 17.4.33 nth\_element()

- Header files:

```
#include <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 underlying datatype is used.
- 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:

```

#include <algorithm>
#include <iostream>
#include <functional>

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] << endl;
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    nth_element(ia, ia + 5, ia + 10, greater<int>());

    cout << "sorting with respect to " << ia[5] << endl;
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;
}
/*

```

```

        generated output:
    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
    */

```

### 17.4.34 `partial_sort()`

- Header files:

```
#include <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 `middle - first` smallest elements are sorted and stored in the `[first, middle)`, using the operator `<()` of the underlying datatype. The remaining elements of the series remain unsorted, and are stored in `[middle, last)`.
- The second prototype: The `middle - first` smallest elements (according to the provided binary predicate `comp`) are sorted and stored in the `[first, middle)`. The remaining elements of the series remain unsorted.

- Example:

```

#include <algorithm>
#include <iostream>
#include <functional>

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:\n";
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "find the 5 biggest elements:\n";
    partial_sort(ia, ia + 5, ia + 10, greater<int>());
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;
}
/*
    generated output:
    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
*/

```

### 17.4.35 `partial_sort_copy()`

- Header files:

```
#include <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 smallest elements in the range `[first, last)` are copied to the range `[dest_first, dest_last)`, using the operator`<()` of the underlying datatype. Only the number of elements in the smaller range are copied to the second range.
- The second prototype: The elements in the range `[first, last)` are sorted by the binary predicate `comp`. The elements for which the predicate returns most often `true` are copied to the range `[dest_first, dest_last)`. Only the number of elements in the smaller range are copied to the second range.

- Example:

```

#include <algorithm>
#include <iostream>
#include <functional>

int main()
{
    int
        ia[] = {1, 10, 3, 8, 5, 6, 7, 4, 9, 2},
        ia2[6];

    partial_sort_copy(ia, ia + 10, ia2, ia2 + 6);

    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;
    cout << "the 6 smallest elements: ";
    copy(ia2, ia2 + 6, ostream_iterator<int>(cout, " "));
    cout << endl;

    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 << endl;

    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 << endl;
}
/*
    generated output:
    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
*/

```

### 17.4.36 partial\_sum()

- Header files:

```
#include <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: the value of each element in the range `[result, <returned OutputIterator>)` 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:

```

#include <numeric>
#include <algorithm>
#include <iostream>
#include <functional>

int main()
{
    int
        ia[] = {1, 2, 3, 4, 5},
        ia2[5];

    copy(ia2,
        partial_sum(ia, ia + 5, ia2),
        ostream_iterator<int>(cout, " "));
}

```

```

        cout << endl;

        copy(ia2,
            partial_sum(ia, ia + 5, ia2, multiplies<int>()),
            ostream_iterator<int>(cout, " "));
        cout << endl;
    }
    /*
        generated output:
        1 3 6 10 15
        1 2 6 24 120
    */

```

### 17.4.37 partition()

- Header files:

```
#include <algorithm>
```

- Function prototypes:

```
- 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>

class LessThan
{
public:
    LessThan(int x): x(x)
    {}
    bool operator()(int value)
    {
        return (value <= x);
    }
private:
    int
        x;
};

int main()
{
    int
        ia[] = {1, 3, 5, 7, 9, 10, 2, 8, 6, 4},
        *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 << endl;
}
/*
    generated output:
    Last element <= 4 is ia[3]
    1 3 4 2 9 10 7 8 6 5
*/

```

### 17.4.38 prev\_permutation()

- Header files:

```
#include <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 in such a way that the first ordering is obtained representing a 'smaller' value (see next\_permutation() (section 17.4.32) 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 underlying data type.
- The second prototype: The previous permutation given the sequence of elements in the range [first, last) is determined. 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:

```

#include <algorithm>
#include <iostream>
#include <string>

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 << endl;
    }
    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 << endl;
    }
    cout << "No (more) previous permutations\n";
}
/*
    generated output:
    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
*/

```

### 17.4.39 random\_shuffle()

- Header files:

```
#include <algorithm>
```

- Function prototypes:

```
- void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
```

```
- void random_shuffle(RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator& rand);
```

- Description:

- The first prototype: The elements in the range [first, last) are randomly reordered.
- The second prototype: The elements in the range [first, last) are randomly reordered, using the rand random number generator, which should return an int in the range [0, remaining), where remaining is passed as argument to the operator()() of the rand function object.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <time.h>

class randomGenerator
{
public:
    randomGenerator()
    {
        srand(static_cast<int>(time(0)));
    }
    int operator()(int remaining) const
    {
        return (rand() % remaining);
    }
};

int main()
{
    string
        words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa"};
    unsigned
        size = sizeof(words) / sizeof(string);

    random_shuffle(words, words + size);

    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    cout << "sorting the words again\n";
    sort(words, words + size);

    randomGenerator
        rg;
    random_shuffle(words, words + size, rg);

    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
```

```

    generated output:
lima oscar mike november papa kilo
sorting the words again
papa mike oscar kilo lima november
*/

```

#### 17.4.40 remove()

- Header file:

```
#include <algorithm>
```

- Function prototype:

```
- ForwardIterator remove(ForwardIterator first, ForwardIterator last, Type &value);
```

- Description:

- The elements in the range pointed to by [first, last) are reordered in such a way that all values unequal to value are placed at the beginning of the range. The returned forward iterator points to the first element, after reordering, that can be removed. The range [return value, last) is called the *leftover* of the algorithm. Note that the leftover may contain other values than value, but these can also safely be removed, as they are also present in the range [first, return value).

- Example:

```

#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    string
        words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
          "oscar", "alpha", "alpha", "papa", "quebec" },
        *removed;
    unsigned
        size = sizeof(words) / sizeof(string);

    cout << "Removing all \"alpha\"s:\n";
    removed = remove(words, words + size, "alpha");
    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
        << "Trailing elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*

```

```

    generated output:
Removing all "alpha"s:
kilo lima mike november oscar papa quebec
Trailing elements are:
oscar alpha alpha papa quebec

```

\*/

### 17.4.41 remove\_copy()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

- `OutputIterator remove_copy(InputIterator first, InputIterator last, OutputIterator result, Type &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.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>

int main()
{
    string
        words[] =
            { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
              "oscar", "alpha", "alpha", "papa", "quebec" };
    unsigned
        size = sizeof(words) / sizeof(string);
    string
        *returnvalue,
        remaining[size - count_if(words, words + size,
                                   bind2nd(equal_to<string>(), string("alpha")))];

    returnvalue = remove_copy(words, words + size, remaining, "alpha");

    cout << "Removing all \"alpha\"s:\n";
    copy(remaining, returnvalue, ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
    Removing all "alpha"s:
    kilo lima mike november oscar papa quebec
*/
```

## 17.4.42 remove\_if()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

```
– ForwardIterator remove_if(ForwardIterator first, ForwardIterator last, UnaryPredicate  
pred);
```

- Description:

– The elements in the range pointed to by [first, last) are reordered in such a way that all values for which the unary predicate 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 pred returns true. The range [returnvalue, last) is called the *leftover* of the algorithm. The leftover may contain other values than value, but these can also safely be removed, as they are also present in the range [first, returnvalue).

- Example:

```
#include <functional>
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    string
        words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
          "oscar", "alpha", "alpha", "papa", "quebec" },
        *removed;
    unsigned
        size = sizeof(words) / sizeof(string);

    cout << "Removing all \"alpha\"s:\n";
    removed = remove_if(words, words + size,
        bind2nd(equal_to<string>(), string("alpha")));

    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
        << "Trailing elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
    Removing all "alpha"s:
    kilo lima mike november oscar papa quebec
    Trailing elements are:
    oscar alpha alpha papa quebec
*/
```



### 17.4.43 remove\_copy\_if()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

```
- 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 copied to the range [result, returnvalue), where returnvalue is the value returned by the function. The range [first, last) is not modified.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>

int main()
{
    string
        words[] =
            { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
              "oscar", "alpha", "alpha", "papa", "quebec" };
    unsigned
        size = sizeof(words) / sizeof(string);
    string
        *returnvalue,
        remaining[size - count_if(words, words + size,
                                   bind2nd(equal_to<string>(), "alpha"))];

    returnvalue = remove_copy_if(words, words + size, remaining,
                                   bind2nd(equal_to<string>(), "alpha"));

    cout << "Removing all \"alpha\"s:\n";
    copy(remaining, returnvalue, ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
    Removing all "alpha"s:
    kilo lima mike november oscar papa quebec
*/
```

### 17.4.44 replace()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

- `ForwardIterator replace(ForwardIterator first, ForwardIterator last, Type &oldvalue, Type &newvalue);`

- Description:

- All elements equal to `oldvalue` in the range pointed to by `[first, last)` are replaced by the value `newvalue`.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    string
        words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
          "oscar", "alpha", "alpha", "papa", "quebec" },
        *removed;
    unsigned
        size = sizeof(words) / sizeof(string);

    replace(words, words + size, string("alpha"), string("ALPHA"));
    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
    kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA ALPHA papa quebec
*/
```

#### 17.4.45 `replace_copy()`

- Header file:

```
#include <algorithm>
```

- Function prototypes:

- `OutputIterator replace_copy(InputIterator first, InputIterator last, OutputIterator result, Type &oldvalue, Type &newvalue);`

- Description:

- All elements equal to `oldvalue` in the range pointed to by `[first, last)` are replaced by the value `newvalue` in a new range `[result, returnvalue)`, where `returnvalue` is the return value of the function.

- Example:

```
#include <algorithm>
#include <iostream>
```

```

#include <string>

int main()
{
    string
        words[] =
            { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
              "oscar", "alpha", "alpha", "papa", "quebec" };
    unsigned
        size = sizeof(words) / sizeof(string);
    string
        remaining[size],
        *returnvalue;

    returnvalue = replace_copy(words, words + size, remaining,
                               string("alpha"), string("ALPHA"));

    copy(remaining, returnvalue, ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
    kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA ALPHA papa quebec
*/

```

#### 17.4.46 replace\_if()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

```
- ForwardIterator replace_if(ForwardIterator first, ForwardIterator last, UnaryPredicate
    pred, Type const &value);
```

- Description:

```
- The elements in the range pointed to by [first, last) for which the unary predicate
    pred evaluates as true are replaced by newvalue.
```

Example:

```

#include <algorithm>
#include <iostream>
#include <string>
#include <functional>

int main()
{
    string
        words[] =
            { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
              "oscar", "alpha", "alpha", "papa", "quebec" };
    unsigned

```

```

        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 << endl;
    }
    /*
        generated output:
        kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA ALPHA papa quebec
    */

```

#### 17.4.47 replace\_copy\_if()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

```
- OutputIterator replace_copy_if(ForwardIterator first, ForwardIterator last, OutputIterator
    result, UnaryPredicate pred, Type const &value);
```

- Description:

```
- The elements in the range pointed to by [first, last) are copied to the range [result,
    returnvalue), where returnvalue is the value returned by the function. The elements
    for which the unary predicate pred returns true are replaced by newvalue. The range
    [first, last) is not modified.
```

- Example:

```

#include <algorithm>
#include <iostream>
#include <string>
#include <functional>

int main()
{
    string
        words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
          "oscar", "alpha", "alpha", "papa", "quebec" };
    unsigned
        size = sizeof(words) / sizeof(string);
    string
        result[size];

    replace_copy_if(words, words + size, result,
                    bind1st(greater<string>(), string("mike")),
                    string("ALPHA"));
    copy (result, result + size, ostream_iterator<string>(cout, " "));
    cout << endl;
}

```

```

/*
    generated output:
    ALPHA ALPHA ALPHA mike ALPHA november ALPHA oscar ALPHA ALPHA papa quebec
*/

```

#### 17.4.48 reverse()

- Header files:

```
#include <algorithm>
```

- Function prototypes:

```
- void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

- Description:

```
- The elements in the range pointed to by [first, last) are reversed.
```

- Example:

```

#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    string
        line;

    while (getline(cin, line))
    {
        reverse(line.begin(), line.end());
        cout << line << endl;
    }
}

```

#### 17.4.49 reverse\_copy()

- Header files:

```
#include <algorithm>
```

- Function prototypes:

```
- 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:

```
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    string
        line;

    while (getline(cin, line))
    {
        unsigned
            size = line.size();
        char
            copy[size + 1];

        cout << "line: " << line << endl <<
            "reversed: ";
        reverse_copy(line.begin(), line.end(), copy);
        copy[size] = 0;      // 0 is not part of the reversed
                            // line !
        cout << copy << endl;
    }
}
```

#### 17.4.50 rotate()

- Header files:

```
#include <algorithm>
```

- Function prototypes:

```
- void rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);
```

- Description:

- The elements implied by the range [first, middle) are moved to the end of the container, the elements implied by the range [middle, last) are moved to the beginning of the container.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    string
        words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa",
```

```

        "echo", "foxtrot", "golf", "hotel", "india", "juliet" };
unsigned const
    size = sizeof(words) / sizeof(string),
    midsize = 6;

rotate(words, words + midsize, words + size);

copy(words, words + size, ostream_iterator<string>(cout, " "));
cout << endl;
}
/*
    generated output:
echo foxtrot golf hotel india juliet kilo lima mike november oscar papa
*/

```

### 17.4.51 rotate\_copy()

- Header files:

```
#include <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 the
    range [first, middle) are copied to the destination container having range [result,
    returnvalue), where returnvalue is the iterator returned by the function.
```

- Example:

```

#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    string
        words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa",
          "echo", "foxtrot", "golf", "hotel", "india", "juliet" };
    unsigned const
        size = sizeof(words) / sizeof(string),
        midsize = 6;
    string
        out[size];

    copy(out,
        rotate_copy(words, words + midsize, words + size, out),
        ostream_iterator<string>(cout, " "));
    cout << endl;
}

```

```

/*
    generated output:
    echo foxtrot golf hotel india juliet kilo lima mike november oscar papa
*/

```

### 17.4.52 search()

- Header files:

```
#include <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 operator==( ) operator of the underlying data type. 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 pred to compare the elements in the two ranges. If no such location exists, last1 is returned.

- Example:

```

#include <algorithm>
#include <iostream>

class absInt
{
public:
    bool operator()(int i1, int i2)
    {
        return (abs(i1) == abs(i2));
    }
};

int main()
{
    int
        range1[] =
            {-2, -4, -6, -8, 2, 4, 6, 8},
        range2[] =
            {6, 8};

    copy
    (
        search(range1, range1 + 8, range2, range2 + 2),

```



```

        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;

    copy
    (
        search(range1, range1 + 8, range2, range2 + 2, absInt()),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;
}
/*
    generated output:
    6 8
    -6 -8 2 4 6 8
*/

```

### 17.4.53 search\_n()

- Header files:

```
#include <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` elements having value `value` are found, using `operator==()` operator of the underlying data type 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` 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:

```

#include <algorithm>
#include <iostream>

class absInt
{
public:
    bool operator()(int i1, int i2)
    {
        return (abs(i1) == abs(i2));
    }
};

```

```

int main()
{
    int
        range1[] =
            {-2, -4, -4, -6, -8, 2, 4, 4, 6, 8},
        range2[] =
            {6, 8};

    copy
    (
        search_n(range1, range1 + 8, 2, 4),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;

    copy
    (
        search_n(range1, range1 + 8, 2, 4, absInt()),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;
}
/*
    generated output:
    4 4
    -4 -4 -6 -8 2 4 4
*/

```

#### 17.4.54 set\_difference()

- Header files:

```
#include <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` that is returned by the function. The elements in the two ranges must have been sorted using `operator<()` of the underlying datatype.
- 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` that is returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>

class CaseLess
{
public:
    bool operator()(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" },
        set2[] =
            { "papa", "quebec", "romeo"},
        result[7],
        *returned;

    copy(result,
        set_difference(set1, set1 + 7, set2, set2 + 3, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string
        set3[] =
            { "PAPA", "QUEBEC", "ROMEO"};
    copy(result,
        set_difference(set1, set1 + 7, set3, set3 + 3, result,
            CaseLess()),
        ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
    kilo lima mike november oscar
    kilo lima mike november oscar
*/
```

#### 17.4.55 set\_intersection()

- Header files:

```
#include <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 `[first1, last1)` that are also present in the ranges `[first2, last2)` is returned, starting at `[result)`, and ending at the `OutputIterator` that is returned by the function. The elements in the two ranges must have been sorted using `operator<()` of the underlying datatype.
- The second prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are also present in the ranges `[first2, last2)` is returned, starting at `[result)`, and ending at the `OutputIterator` that is returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>

class CaseLess
{
public:
    bool operator()(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" },
        set2[] =
        { "papa", "quebec", "romeo"},
        result[7],
        *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;
```

```

}
/*
    generated output:
papa quebec
papa quebec
*/

```

### 17.4.56 set\_symmetric\_difference()

- Header files:

```
#include <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` that is returned by the function. The elements in the two ranges must have been sorted using `operator<()` of the underlying datatype.
- The second prototype: a sorted sequence of the elements 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` that is returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.

- Example:

```

#include <algorithm>
#include <iostream>
#include <string>

class CaseLess
{
public:
    bool operator()(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" },
    set2[] =
    { "papa", "quebec", "romeo"},
    result[7],
    *returned;

    copy(result,
        set_symmetric_difference(set1, set1 + 7, set2, set2 + 3, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string
        set3[] =
        { "PAPA", "QUEBEC", "ROMEO"};
    copy(result,
        set_symmetric_difference(set1, set1 + 7, set3, set3 + 3, result,
            CaseLess()),
        ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
    kilo lima mike november oscar romeo
    kilo lima mike november oscar ROMEO
*/

```

### 17.4.57 set\_union()

- Header files:

```
#include <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 `[first1, last1)` that are also present in the ranges `[first2, last2)` is returned, starting at `[result)`, and ending at the `OutputIterator` that is returned by the function. The elements in the two ranges must have been sorted using `operator<()` of the underlying datatype.
- The second prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are also present in the ranges `[first2, last2)` is returned, starting at `[result)`, and ending at the `OutputIterator` that is returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.

- Example:

```

#include <algorithm>
#include <iostream>

```

```

#include <string>

class CaseLess
{
public:
    bool operator()(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" },
        set2[] =
        { "papa", "quebec", "romeo"},
        result[7],
        *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;
}
/*
    generated output:
    papa quebec
    papa quebec
*/

```

### 17.4.58 sort()

- Header files:

```
#include <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 underlying datatype.
  - The second prototype: the elements in the range `[first, last)` are sorted in ascending order, using the `comp` function object to compare the elements.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>

int main()
{
    string
        words[] =
            {"november", "kilo", "mike", "lima",
             "oscar", "quebec", "papa"};

    sort(words, words + 7);
    copy(words, words + 7, ostream_iterator<string>(cout, " "));
    cout << endl;

    sort(words, words + 7, greater<string>());
    copy(words, words + 7, ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
    kilo lima mike november oscar papa quebec
    quebec papa oscar november mike lima kilo
*/
```

#### 17.4.59 `stable_partition()`

- Header files:
 

```
#include <algorithm>
```
- Function prototypes:
  - `BidirectionalIterator stable_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 relative order of the elements in the container is kept. 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 <functional>

int main()
{
    int
        org[] = {1, 3, 5, 7, 9, 10, 2, 8, 6, 4},
        ia[10],
        *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 << endl;

    copy(org, org + 10, ia);
    split = stable_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 << endl;
}
/*
    generated output:
    Last element <= 4 is ia[3]
    1 3 4 2 9 10 7 8 6 5
    Last element <= 4 is ia[3]
    1 3 2 4 5 7 9 10 8 6
*/
```

#### 17.4.60 `stable_sort()`

- Header files:

```
#include <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 underlying datatype: the relative order of the equal elements is kept.
```

- The second prototype: the elements in the range [first, last) are stable-sorted in ascending order, using the comp function object to compare the elements.

- Example (annotated below):

```

#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <vector>

typedef pair<string, string>    pss;                // 1 (see the text)

class sortby
{
public:
    sortby(string pss::*field)                    // 2
    :
        field(field)
    {}

    bool operator()(pss const &p1, pss const &p2) const    // 3
    {
        return p1.*field < p2.*field;
    }
private:
    string
        pss::*field;
};

ostream &operator<<(ostream &out, pss const &p)          // 4
{
    return out << "    " << p.first << " " << p.second << endl;
}

int main()
{
    vector<pss>                                // 5
        namecity;

    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:\n";
    copy(namecity.begin(), namecity.end(), ostream_iterator<pss>(cout));

                                                                    // 7
    stable_sort(namecity.begin(), namecity.end(), sortby(&pss::second));

```

```

        cout << "sorted by names within sorted cities:\n";
        copy(namecity.begin(), namecity.end(), ostream_iterator<pss>(cout));
    }
    /*
        generated output:
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 by multiple hierarchical criteria. The example deserves some extra attention:

1. First, a typedef is used to reduce the clutter that occur from the repeated use of `pair<string, string>`.
2. Then, a class `sortby` is defined, allowing us to construct an anonymous object which receives a pointer to one of the pair data members that are used for sorting. In this case, as both members are string objects, the constructor can easily be defined: its parameter is a pointer to a string member of the class `pair<string, string>`.
3. The `operator()()` member will receive two pair references, and it will then use the pointer to its members, stored in the `sortby` object, to compare the appropriate fields of the pairs.
4. Then, `operator<<()` is overloaded to be able to insert a pair into an `ostream` object. This is merely a service function to make life easy.
5. In `main()`, first some data is stored in a vector.
6. Then the first sorting takes place. The least important criterion must be sorted first, and for this a simple `sort()` will suffice. 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)`.
7. The next important criterion, the cities, are sorted next. Since the relative ordering of the *names* will not be 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 will not be 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)`.

### 17.4.61 swap()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

- void swap(Type &object1, Type &object2);

- Description:

- The elements object1 and object2 change their values.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    string
        first[] = {"alpha", "bravo", "charley"},
        second[] = {"echo", "foxtrot", "golf"};
    unsigned
        n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    for (unsigned idx = 0; idx < n; ++idx)
        swap(first[idx], second[idx]);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
Before:
alpha bravo charley
echo foxtrot golf
After:
echo foxtrot golf
alpha bravo charley
*/
```

### 17.4.62 swap\_ranges()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

- `ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 result);`

- Description:

- The elements in the ranges pointed to by `[first1, last1)` are swapped with the elements in the ranges `[result, returnvalue)`, where `returnvalue` is the value returned by the function. The two ranges must be disjoint.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    string
        first[] = {"alpha", "bravo", "charley"},
        second[] = {"echo", "foxtrot", "golf"};
    unsigned
        n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    swap_ranges(first, first + n, second);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
Before:
alpha bravo charley
echo foxtrot golf
After:
echo foxtrot golf
alpha bravo charley
*/
```

### 17.4.63 transform()

- Header files:

```
#include <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 `[first, last)` 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:

```
#include <functional>
#include <vector>
#include <algorithm>
#include <iostream>
#include <string>
#include <cctype>

class Caps
{
public:
    string operator()(string const &src)
    {
        string
            tmp = src;

        transform(tmp.begin(), tmp.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 << endl;

    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 << endl;
}
/*
    generated output:
    ALPHA BRAVO CHARLEY
    1 4 9 16 25
*/

```

### 17.4.64 unique()

- Header file:

```
#include <algorithm>
```

- Function prototypes:

```

- ForwardIterator unique(ForwardIterator first, ForwardIterator last);
- ForwardIterator unique(ForwardIterator first, ForwardIterator last, BinaryPredicate
  pred);

```

- Description:

```

- The first prototype: Consecutively equal elements (using operator==( ) of the underlying
  data type) in the range pointed to by [first, last) are collapsed into a single element.
  The returned forward iterator marks the leftover of the algorithm, and contains (unique)
  elements appearing earlier in the range.
- The second prototype: Consecutive elements in the range pointed to by [first, last)
  for which the binary predicate pred returns true are collapsed into a single element.
  The returned forward iterator marks the leftover of the algorithm, and contains (unique)
  elements appearing earlier in the range.

```

- Example:

```

#include <algorithm>
#include <iostream>
#include <string>

class CaseString
{
public:
    bool operator()(string const &first, string const &second) const
    {
        return (!strcasecmp(first.c_str(), second.c_str()));
    }
};

int main()
{
    string
        words[] =
        {"alpha", "alpha", "Alpha", "papa", "quebec" },

```

```

        *removed;
    unsigned
        size = sizeof(words) / sizeof(string);

    removed = unique(words, words + size);
    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
        << "Trailing elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    removed = unique(words, words + size, CaseString());
    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
        << "Trailing elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
alpha Alpha papa quebec
Trailing elements are:
quebec
alpha papa quebec
Trailing elements are:
quebec quebec
*/

```

### 17.4.65 unique\_copy()

- Header file:

```
#include <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 underlying data type) are copied only once. The returned output iterator points just beyond the last element that was copied.
- 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. The returned output iterator points just beyond the last element that was copied.

- Example:

```
#include <algorithm>
```



```

#include <iostream>
#include <string>
#include <vector>
#include <functional>

class CaseString
{
public:
    bool operator()(string const &first, string const &second) const
    {
        return (!strcasecmp(first.c_str(), second.c_str()));
    }
};

int main()
{
    string
        words[] = {"oscar", "Alpha", "alpha", "alpha", "papa", "quebec" };
    unsigned
        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 << endl;

    vector<string>
        remaining2;

    unique_copy(words, words + size,
                back_inserter(remaining2), CaseString());

    copy(remaining2.begin(), remaining2.end(),
        ostream_iterator<string>(cout, " "));
    cout << endl;
}
/*
    generated output:
    oscar Alpha alpha papa quebec
    oscar Alpha papa quebec
*/

```

#### 17.4.66 upper\_bound()

- Header files:

```
#include <algorithm>
```

- Function prototypes:

- `ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const Type &value);`
- `ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const Type &value, Compare comp);`

- Description:

- The first prototype: The sorted elements stored in the iterator range `[first, last)` are searched for the first element that is greater than `value`. The returned iterator marks the location in the sequence where `value` can be inserted without breaking the sorted order of the elements, using `operator<()` of the underlying datatype. 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 (-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:

```
#include <algorithm>
#include <iostream>
#include <functional>

int main()
{
    int
        ia[] = {10, 15, 15, 20, 30};
    unsigned
        n = sizeof(ia) / sizeof(int);

    cout << "Sequence: ";
    copy(ia, ia + n, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "15 can be inserted before " <<
        *upper_bound(ia, ia + n, 15) << endl;
    cout << "35 can be inserted after " <<
        (upper_bound(ia, ia + n, 35) == ia + n ?
         "the last element" : "??") << endl;

    sort(ia, ia + n, greater<int>());

    cout << "Sequence: ";
    copy(ia, ia + n, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "15 can be inserted before " <<
        *upper_bound(ia, ia + n, 15, greater<int>()) << endl;
    cout << "35 can be inserted before " <<
        (upper_bound(ia, ia + n, 35, greater<int>()) == ia ?
         "the first element" : "??") << endl;
}
/*
generated output:
```

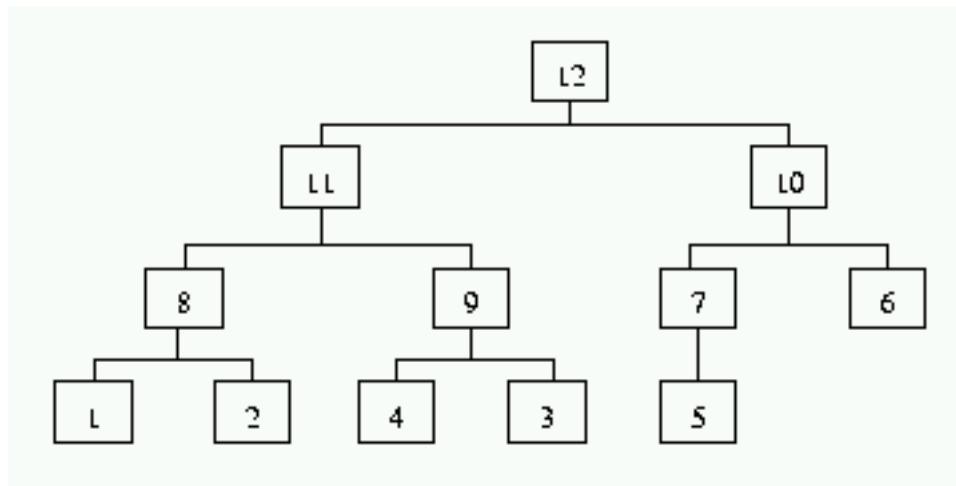


Figure 17.1: A binary tree representation of a heap

```

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
*/

```

#### 17.4.67 Heap algorithms

A heap is a form of binary tree represented as an array. In the standard heap, the key of an element is greater or equal to 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 follows:

This tree can be organized in an array as follows:

12, 11, 10, 8, 9, 7, 6, 1, 2, 4, 3, 5

Here, 12 is the top node. its children are 11 and 10, both less than 12. 11, in turn, has 8 and 9 as its children, while the children of 10 are 7 and 6. 8 has 1 and 2 as its children, 9 has 4 and 3, and finally, 7 has left child 5. 7 doesn't have a right child, and 6 has no children.

Note that the left and right branches are not ordered: 8 is less than 9, but 7 is larger than 6.

The heap is formed 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 6, 7, 8 and 9 are found, etc.

Heaps can be created in containers supporting random access. So, a heap is not, for example, constructed in a list. 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 invalidates the heap, though).

The following subsections introduce the prototypes of the heap-algorithms, the final subsection provides a small example in which the heap algorithms are used.

### **make\_heap()**

- Header files:

```
#include <algorithm>
```

- Function prototypes:

```
- void make_heap(RandomAccessIterator first, RandomAccessIterator last);  
- void make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

- Description:

- The first prototype: The elements in the range `[first, last)` are reordered to form a max-heap, using `operator<()` of the underlying data type.
- The second prototype: The elements in the range `[first, last)` are reordered to form a heap, using the binary comparison function object `comp` to compare elements.

- Follow this link for a small example of a program using `make_heap()`.

### **pop\_heap()**

- Header files:

```
#include <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 `[first, last)` is moved to `last - 1`. Then, the elements in the range `[first, last - 1)` are reordered to form a max-heap, using the `operator<()` of the underlying data type.
- The second prototype: The first element in the range `[first, last)` is moved to `last - 1`. Then, the elements in the range `[first, last - 1)` are reordered to form a heap, using the binary comparison function object `comp` to compare elements.

- Follow this link for a small example of a program using `pop_heap()`.

### **push\_heap()**

- Header files:

```
#include <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 `[first, last - 2)` contains a valid heap, and the element at `last - 1` contains an element to be added to the heap, the elements in the range `[first, last - 1)` are reordered to form a max-heap, using the `operator<()` of the underlying data type.
  - The second prototype: Assuming that the range `[first, last - 2)` contains a valid heap, and the element at `last - 1` contains an element to be added to the heap, the elements in the range `[first, last - 1)` are reordered to form a heap, using the binary comparison function object `comp` to compare elements.
- Follow this link for a small example of a program using `push_heap()`.

## sort\_heap()

- Header files:
 

```
#include <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 underlying data type.
  - 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.
- Follow this link for a small example of a program using `sort_heap()`.

## An example using the heap algorithms

```
#include <algorithm>
#include <iostream>
#include <functional>

void show(int *ia, char const *header)
{
    cout << header << ":\n";
    copy(ia, ia + 20, ostream_iterator<int>(cout, " "));
    cout << endl;
}
```

```

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, "Adding 20 (at the end) to the heap again");

    sort_heap(ia, ia + 20, greater<int>());
    show(ia, "Sorting the elements in the heap");
}
/*
    generated output:
The values 1-20 in a max-heap:
20 19 15 18 11 13 14 17 9 10 2 12 6 3 7 16 8 4 1 5
Removing the first element (now at the end):
19 18 15 17 11 13 14 16 9 10 2 12 6 3 7 5 8 4 1 20
Adding 20 (at the end) to the heap again:
20 19 15 17 18 13 14 16 9 11 2 12 6 3 7 5 8 4 1 10
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
Adding 20 (at the end) to the heap again:
1 2 3 8 4 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
*/

```

# Chapter 18

## Templates

The C++ language support a mechanism which allows programmers to define completely general functions or classes, based on hypothetical arguments or other entities. Code in which this mechanism has been used is found in, e.g., the chapter on abstract containers.

These general functions or classes are used to generate concrete code once their definitions are applied to real entities. Such a general definition of a function or a class is called a *template*, the concrete code that is generated based on a template is called an *instantiation*.

In this chapter we will examine template functions and template classes.

### 18.1 Template functions

Template functions are used in cases where a single implementation of a function is not practical due to the different types that are distinguished in C++. If a function is declared as

```
fun(int *array){
```

then this function will likely run into problems if it is passed the address of an array of double values. The function will normally have to be duplicated for parameters of different types. For example, a function computing the sum of the elements of an array for an array of ints is:

```
int sumVector(int *array, unsigned n)
{
    int
        sum(0);
    for (int idx = 0; idx < n; ++idx)
        sum += array[idx];
    return (sum);
}
```

The function must be overloaded for arrays of doubles:

```
double sumVector(double *array, unsigned n)
{
```

```

double
    sum(0);
for (int idx = 0; idx < n; ++idx)
    sum += array[idx];
return (sum);
}

```

In a local program development situation this hardly ever happens, since only one or two `sumVector()` implementations will be required. But the strongly typed nature of C++ stands in the way of creating a truly general function, that can be used for any type of array.

In cases like these, *template functions* are used to create the truly general function. The template function can be considered a general *recipe* for constructing a function that can be used with the generic array. In the upcoming sections we'll discuss the construction of template functions. First, the construction of a template function is discussed. Then the instantiation is covered. With template functions the argument deduction deserves special attention. See section 18.1.3 for this topic.

### 18.1.1 Template function definitions

The definition of a template function is very similar to the definition of a normal function, except for the fact that the parameters, the types that are used in the function, and the function's return value may be specified in a completely general way. The function `sumVector()` in the previous section can be rewritten as a template function:

```

#include <numeric>

template <typename T>
T sumVector(T *array, unsigned n)
{
    return accumulate(array, array + n, T());
}

```

Note the correspondence with the formerly defined `sumVector()` functions. In fact, if a `typedef int T` had been specified, the template function, except for the initial `template` line, *would* be the first `sumVector()` function of the previous section. So, the essence of the template function is found in the first line. From the above example:

```
template <typename T>
```

Definitions or declarations of templates always start with the `template <...>` line. It is followed by the *template parameter list*, which is a comma-separated non-empty list of so-called *template type* or *template non-type parameters*, surrounded by angular brackets `<` and `>`. In the Annotations template type parameters always start with the keyword `typename`. In other texts on C++ the keyword `class` can also be encountered. So, in other texts template definitions might start with a line like:

```
template <class T>
```

Using `class` instead of `typename` is now, however, considered an anachronism, and is deprecated: a template type parameter is, after all, a type name and not a class.



In the template function `sumVector()` the only template parameter is `T`, which is a template type parameter. `T` is the formal type that is used in the template function definition to represent the actual type that will be specified when the template function is instantiated. This type is used in the parameter list of the function, it is used to define the type of a local variable of the function, and it is used to define the return type of the function.

Normal scope rules and identifier rules apply to template definitions and declarations: the type `T` is a *formal name*, it could have been named `Type`. The formal typename that is used overrules, within the scope of the template definition or declaration, any previously defined identifiers by that name.

A template *non-type parameter* represents 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 an unsigned. An alternative definition for the above template function, using a template non-type parameter is (note that there are two versions: one for arrays having `const` elements, one for arrays having elements that can be modified):

```
#include <numeric>

template <typename T, unsigned size>      // non-const arrays
T sumVector(T (&array)[size])
{
    return accumulate(array, array + size, T());
}

template <typename T, unsigned size>      // const arrays
T sumVector(const T (&array)[size])
{
    return accumulate(array, array + size, T());
}
```

Template function definitions may have multiple type and non-type parameters. Each parameter name must be unique. For example, the following template defines a template function for a function `outerProduct()`, returning a pointer to vectors of `size2` `T2` elements, and expecting two vectors of, respectively, `size1` and `size2` elements:

```
template
<
    class T1,
    class T2,
    unsigned size1,
    unsigned size2
>
T1
(
    *outerProduct
    (
        T2 const (&v1)[size1],
        T2 const (&v2)[size2]
    )
)[size2];
```

Note that the return type `T1` of the returned vectors is intentionally specified different from `T2`. This allows us to specify, e.g., return type `double` for the returned outer product, while the vectors passed to `outerProduct` are of type `int`.

The keyword `typename` is *required* in certain situations that may occur when the template function is defined. For example, assume we define the following template function:

```
template <typename T>
void function()
{
    T::member
        *p;
}
```

Although the layout of the above function suggests that `p` is defined as a pointer to the type `member`, that must have been declared in the class that is specified when the function is instantiated, it actually is interpreted by the compiler as a multiplication of `T::member` and `p`.

The compiler does so, because it cannot know from the template definition whether `member` is a *typename*, defined in the class `T`, or a *member* of the class `T`. It takes the latter and, consequently, interprets the `*` as a multiplication operator.

What if this interpretation was not intended? In that case the `typename` keyword must be used. In the following template definition the `*` indicates a pointer definition to a `T::member` type.

```
template <typename T>
void function()
{
    typename T::member
        *p;
}
```

## 18.1.2 Instantiations of template functions

Consider the first template function definition in section 18.1.1. This definition is a mere recipe for constructing a particular function. The function is actually constructed once it is used, or its address is taken. Its type is implicitly defined by the nature of its parameters.

For example, in the following code it is assumed that the function `sumVector` has been defined in the header file `sumvector.h`. In the function `main()` the function `sumVector()` is called once for the `int` array `x`, once for the `double` array `y`, and once the address is taken of a `sumVector()` function. By taking the address of a `sumVector` function the type of the argument is defined by the type of the pointer variable, in this case a pointer to a function processing a array of unsigned long values. Since such a function wasn't available yet (we had functions for `ints` and `doubles`), it is constructed once its address is required. Here is the function `main()`:

```
#include <iostream>
#include "sumvector.h"

int main()
{
    int
        x[] = {1, 2};
    double
        y[] = {1.1, 2.2};

    cout << sumVector(x, 2) << endl    // first instantiation
```

```

        << sumVector(y, 2) << endl;    // second instantiation

    unsigned long                // third instantiation
        (*pf)(unsigned long *, unsigned) = sumVector;
}
/*
    generated output:
3
3.3
*/

```

While in the above example the functions `sumVector()` could be instantiated, this is not always possible. Consider the following code:

```

#include "sumvector.h"

unsigned fun(unsigned (*f)(unsigned *p, unsigned n));
double fun(double (*f)(double *p, unsigned n));

int main()
{
    cout << fun(sumVector) << endl;
}

```

In the above example the function `fun()` is called in the function `main()`. Although it appears that the address of the function `sumVector()` is passed over to the function `fun()`, there is a slight problem: there are two overloaded versions of the function `fun()`, and *both* can be given the address of a function `sumVector()`. The first function `fun()` expects an `unsigned *`, the second one a `double *`. Which instantiation must be used for `sumVector()` in the `fun(sumVector)` expression? This is an ambiguity, which makes the compiler balk. The compiler complains with a message like

```

In function 'int main()':
call of overloaded 'fun ({unknown type})' is ambiguous
candidates are: fun(unsigned int (*)(unsigned int *, unsigned int))
                fun(double (*)(double *, unsigned int))

```

Situations like this should of course be avoided. Template functions can only be instantiated if this can be done unambiguously. It is, however, possible to disambiguate the situation using a type cast. In the following code fragment the (proper) `double *` implementation is forced by means of a `static_cast`:

```

#include "sumvector.h"

unsigned fun(unsigned (*f)(unsigned *p, unsigned n));
double fun(double (*f)(double *p, unsigned n));

int main()
{
    cout << fun(static_cast<double (*)(double *, unsigned)>(sumVector))
        << endl;
}

```

But type casts should be avoided, where possible. Fortunately the cast can be avoided in this kind of situation, as described in section 18.1.4.

If the same template function definition was included in different source files, which are then compiled to different object files which are thereupon linked together, there will, per type of template function, be only one instantiation of the template function in the final program.

This is illustrated by the following example, in which the address of a function `sumVector()` for `int` arrays is written to `cout`. The first part defines a function `fun()` in which the address of a `sumVector()` function is written to `cout`. The second part defines a function `main()`, defined in a different sourcefile, in which the address of a similar `sumVector()` function is written to `cout`, and in which `fun()` is called. Both sources use the union `PointerUnion` to convert a pointer to a function to a `void *`. Here is the used union:

```
union PointerUnion
{
    int (*fp)(int *, unsigned);
    void *vp;
};
```

The source of the function `fun()`:

```
#include "sumvector.h"
#include "pointerunion.h"

void fun()
{
    PointerUnion
        pu = { sumVector };

    cout << pu.vp << endl;
}
```

And the code of `main()`:

```
#include "sumvector.h"
#include "pointerunion.h"

void fun();

int main()
{
    fun();

    PointerUnion
        pu = { sumVector };

    cout << pu.vp << endl;
}
/*
    generated output:
0x8048710
0x8048710
*/
```

Note that the program shows the same function addresses for the two instantiations: in the final program only one copy of `sumVector()` is used.

## Declaring template functions

Template functions can be *declared* as well. If it is *known* that the required instantiation is available from another source file, as the `sumVector` that was used in the function `fun()` in the previous section, then there is actually no need to instantiate the function `sumVector()` again.

A template function declaration is constructed like any other function declaration: by replacing its code with a semicolon. For example:

```
#include "sumvector.h"
#include "pointerunion.h"

template<typename T>
T sumVector(T *tp, unsigned n);

void fun()
{
    PointerUnion
        pu = { sumVector };

    cout << pu.vp << endl;
}
```

To make this work, one must of course be certain that the instantiation is available elsewhere. The *advantage* of this approach is that the compiler doesn't have to instantiate a template function, which speeds up the compilation of the function in which the template function is only declared. The *disadvantage* is that we have to do the bookkeeping ourselves: is the template function used somewhere else or not?

### 18.1.3 Argument deduction

The compiler determines the type of the template function that must be instantiated by examining the types and values of its arguments. This process is called *template argument deduction*. With template argument deduction, the type of the return value of the template function is *not considered*.

For example, consider once again the function

```
T sumVector(T (&array)[size])
```

given in section 18.1.1:

```
#include <numeric>

template <typename T, unsigned size>          // non-const arrays
T sumVector(T (&array)[size])
{
    return accumulate(array, array + size, T());
}

template <typename T, unsigned size>          // const arrays
T sumVector(const T (&array)[size])
```

```

{
    return accumulate(array, array + size, T());
}

```

In this function the template non-type parameter `size` is determined from the size of the array that is used with the call. Since the size of an array is known to the compiler, the compiler can determine the size parameter by looking up the size of the array that is used as argument to the function `sumVector()`. If the size is not known, e.g., when a pointer to an array element is passed to the function, the compilation will not succeed. Therefore, in the following example, the first call of the function `sumVector()` will succeed, as `iArray` is an array; the second one will fail, as `iPtr` is a pointer, pointing to an array of (in principle) unknown size:

```

#include "sumvectorsize.h"

int main()
{
    int
        iArray[] = {1, 2, 3},
        *iPtr = iArray;

    sumVector(iArray); // succeeds: size of iArray is known
    sumVector(iPtr);   // fails: size of array pointed to by
                        // iPtr is unknown
}

```

It is not necessary for a template function's argument to match exactly the type of the template function's corresponding parameter. Three kinds of conversions are allowed here:

- *lvalue transformations*
- *qualification conversions*
- *conversion to a base class instantiated from a class template*

These three types of conversions are now discussed and illustrated.

## Lvalue transformations

There are three types of *lvalue transformations*:

- *lvalue-to-rvalue conversions*
- *array-to-pointer conversions*
- *function-to-pointer conversions*

**lvalue-to-rvalue conversions.** An lvalue-to-rvalue conversion takes place in situations where the value of an lvalue expression is required. This also happens when a variable is used as argument to a function having a *value parameter*.

**array-to-pointer conversions.** An array-to-pointer conversion occurs when the name of an array is assigned to a pointer variable. This is frequently seen with functions using parameters that are

pointer variables. When calling such functions, an array is often specified as argument to the function. The address of the array is then assigned to the pointer-parameter. This is called an *array-to-pointer conversion*.

**function-to-pointer conversions.** This conversion is most often seen with functions defining a parameter which is a pointer to a function. When calling such a function the name of a function may be specified for the parameter which is a pointer to a function. The address of the function is then assigned to the pointer-parameter. This is called a function-to-pointer conversion.

In the first `sumVector()` template (section 18.1.1) the first parameter is defined as a `T *`. Here an array-to-pointer conversion is allowed, as it is an lvalue transformation, which is one of the three allowed conversions. Therefore, the name of an array may be passed to this function as its first argument.

### Qualification conversions

A *qualification conversion* adds `const` or `volatile` qualifications to *pointers*. Assume the function `sumVector()` in section 18.1.1 was defined as follows:

```
#include <numeric>

template <typename T>
T sumVector(T const *array, unsigned n)
{
    return accumulate(array, array + n, T());
}
```

In the above definition, a plain array or pointer to some type can be used in combination with this function `sumVector()`. E.g., an argument `iArray` could be defined as `int iArray[5]`. However, no damage is inflicted on the elements of `iArray` by the function `sumVector()`: it explicitly states so, by defining `array` as a `T const *`. Qualification conversions are therefore allowed in the process of template argument deduction.

### Conversion to a base class

In section 18.2 template classes are formally introduced. However, they were already *used* earlier: abstract containers (covered in chapter 12) are actually defined as template classes. Like ‘normal’ classes, template classes can participate in the construction of class hierarchies. In section 18.2.7 it is shown how a template class can be derived from another template class.

Assume that the template class `Pipe` is derived from the class `queue`. Furthermore, assume our function `sumVector()` was written to return the sum of the elements of a `queue`:

```
template <typename T>
T sumVector(queue<T> &queue)
{
    T
        sum();

    while (!queue.empty())
    {
```

```

        sum += queue.front();
        queue.pop();
    }
    return (sum);
}

```

All kinds of `queue` objects can be passed to the above function. However, it is also possible to pass `Pipe` objects to the function `sumVector()`: By instantiating the `Pipe` object, its base class, which is the template class `queue`, is also instantiated. Now:

- `Pipe<xxx>` has `queue<xxx>` as its base class, and
- `queue<xxx>` is a possible first argument of the above template function `sumVector()`, and
- a function argument which is of a derived class type may be used with a base class parameter of a template function.

Consequently, the definition '`Pipe<int> pi;`' implies the instantiation of the base class `queue<int>`, which is an allowed type for the first parameter of `sumVector()`. Therefore, `pi` may be passed as argument to `sumVector()`.

This conversion is called a *conversion to a base class instantiated from a class template*. In the above example, the *class template* is `Pipe`, the base class is `queue`.

### Summary: the template argument deduction algorithm

The following algorithm is used with template argument deduction when a template function is called with one or more arguments:

- In turn, the template parameters are identified in the parameters of the called function.
- For each template parameter, the template's type is deduced from the template function's argument (e.g., `int` if the argument is a `Pipe<int>` object).
- The three allowed conversions (see section 18.1.3) for template arguments are applied where necessary.
- If the same template parameter is used with multiple function parameters, the template types of the arguments must be the same. E.g., with template function

```
twoVectors(vector<Type> &v1, vector<Type> &v2)
```

the arguments used with `twoVectors()` must have equal types. E.g.,

```

extern vector<int>
    v1,
    v2;

twoVectors(v1, v2);

```



### 18.1.4 Explicit arguments

Consider once again the function `main()` in section 18.1.2. Here the function `sumVector()` was called as follows:

```
#include <iostream>
#include "sumvector.h"

int main()
{
    int
        x[] = {1, 2};
    double
        y[] = {1.1, 2.2};

    cout << sumVector(x, 2) << endl
         << sumVector(y, 2) << endl;
}
```

In both cases the final argument of the function is of type `int`, but in the template's definition, the second parameter is an `unsigned`. The conversion `unsigned -> int` is not one of the allowed conversions *lvalue transformation*, *qualification conversion* or *conversion to a base class*. Why doesn't the compiler complain in this case? In cases where the type of the argument is fixed, *standard type conversions* are allowed, and they are applied automatically by the compiler. The types of arguments may also be made explicit by providing type casts. In those cases there is no need for the compiler to deduce the types of the arguments.

In section 18.1.2, a type cast was used to disambiguate. Rather than using a `static_cast`, the type of the required function can be made explicit using another syntax: the function name may be followed by the types of the arguments, surrounded by pointed brackets. Here is the example of section 18.1.2 using an *explicit template argument type*:

```
#include <iostream>
#include "sumvector.h"

unsigned fun(unsigned (*f)(unsigned *p, unsigned n));
double fun(double (*f)(double *p, unsigned n));

int main(int argc, char **argv)
{
    cout << fun(sumVector<double>) << endl;
}
```

The explicit argument type list should match the types mentioned in the `template<...>` line preceding the template's function definition. The type `class T` in the template line of the function `sumVector()` is made explicit as type `double`, and not as, e.g., a type `double *`, which was used in the `static_cast` in the example of section 18.1.2.

### Template explicit instantiation declarations

Templates can be *declared*, explicitly providing the types of the template's arguments with the purpose of *instantiating* these templates. Such an *explicit instantiation declaration* starts with the

keyword `template`, to be followed by an explicit template instantiation declaration. Although this is a declaration (see also section 18.1.2), it is also understood by the compiler as a request to instantiate that particular variant of the function.

By using explicit instantiation declarations it is possible to collect all required instantiations of a template functions in one file, so that sources using these template functions can use plain, non-instantiating template function declarations, in order to reduce their compilation time. For example, several `sumVector()` functions can be instantiated as follows:

```
#include "sumvector.h"

template    int sumVector<int>(int *, unsigned);
template  double sumVector<double>(double *, unsigned);
template unsigned sumVector<unsigned>(unsigned *, unsigned);
```

As seen from this example, explicit instantiation declarations are mere function declarations, e.g.,

```
int sumVector(int *, unsigned);
```

embellished with the `template` keyword and an explicit template argument list, e.g., `<int>`.

### 18.1.5 Template explicit specialization

Although the function `sumVector()` we've seen in the previous sections is well suited for arrays of elements of the basic types (like `int`, `double`, etc.), the template implementation is of course not appropriate in cases where the `+=` operator is not defined. In these cases a *template explicit specialization* may be used.

For example, the template's implementation of `sumVector()` is not suited for variables of type `char *`, like the `argv` parameter of `main()`. If we want to be able to use `sumVector()` with variables of type `char *` as well, we can define the following special form of `sumVector()`:

```
#include <iostream>
#include "specialization.h"

int main(int argc, char **argv)
{
    cout << sumVector(argv, argc);
}
```

A template explicit specialization starts with the keyword `template`, followed by an empty set of pointed brackets. This is followed by the head of the function, which must follow the same syntax as a template explicit instantiation declaration, albeit that the trailing `;` of the declaration is replaced by the actual function body of the specialization implementation. In particular note that the ordering of the template parameters and other function parameters in the explicit specialization must be identical to the ordering used in the original template function. So, the template explicit specialization could not use `argc` and `argv` in the order used by, e.g., `main()`.

The template explicit specialization is normally included in the same file as the standard implementation of the template function. If the template explicit specialization is to be used in a different file than the file in which it is defined, it must be declared. Of course, being a template function,

the definition of the template explicit specialization can also be included in every file in which it is used, but that will also slow down the compilation of those other files.

The declaration of a template explicit specialization obeys the standard syntax of a function declaration: the definition is replaced by a semicolon. Therefore, the declaration of the above template explicit specialization is

```
template <> char *sumVector<char *>(char **, unsigned);
```

Note the pair of pointed brackets following the `template` keyword. Were they omitted, the function would reduce to a template instantiation declaration: you would not notice it, except for the longer compilation time, as using a template instantiation declaration implies an extra instantiation (i.e., compilation) of the function. Furthermore, only *one location* of the non-template function would be allowed in the sources of a program, so a header file would not exactly be a good location for the code of a non-template function.

In the *declaration* of the template explicit specialization the explicit specification of the template arguments (in the `< ... >` list following the name of the function) can be omitted if the types of the arguments can be deduced from the types of the arguments. With the above declaration this is the case. Therefore, the declaration can be simplified to:

```
template <> char *sumVector(char **, unsigned);
```

Comparably, the `template <>` part of the template explicit specialization may be omitted. The result is an ordinary function or ordinary function declaration. This is not an error: template functions and non-template functions may overload each other. Ordinary functions are less restrictive in the type conversions that are allowed for their arguments than template functions, which might be a reason for using an ordinary function.

### 18.1.6 Overloading template functions

Template functions may be overloaded. The function `sumVector()` defined earlier (e.g. in section 18.1.1) may be overloaded to accept, e.g., variables of type `vector`:

```
#include <vector>
#include <numeric>

template <typename T>
T sumVector(vector<T> &array)
{
    return (accumulate(array.begin(), array.end(), T()));
}
```

Such a template function can be used by passing it an argument of type `vector`, as in:

```
void fun(vector<int> &vi)
{
    cout << sumVector(vi) << endl;
}
```

Apart from defining overloaded versions, the overloaded versions can of course also be declared. E.g.,

```
template <typename T>
T sumVector(vector<T> &array);
```

Using templates may result in ambiguities which overloading cannot solve. Consider the following template function definition:

```
template<typename T>
bool differentSigns(T v1, T v2)
{
    return
    (
        v1 < 0 && v2 >= 0
        ||
        v1 >= 0 && v2 < 0
    );
}
```

Passing `differentSigns()` an `int` and an `unsigned` is an error, as the two types are different, whereas the template definition calls for identical types. Overloading doesn't really help here: defining a template having the following prototype is ok with the `int` and `unsigned`, but now two instantiations are possible with *identical* types.

```
template<typename T1, typename T2>
bool differentSigns(T1 v1, T2 v2);
```

This situation can be disambiguated by using template explicit arguments, e.g., `differentSigns<int, int>(12, 30)`. But template explicit arguments could be used anyway with the second overloaded version of the function: in that case the first definition is superfluous and could have been omitted.

On the other hand, if one overloaded version can be interpreted as a *more specialized* variant of another version of a template function, then in principle the two variants of the template function could be used if the arguments are of the more specialized types. In this case, there is no ambiguity, as the compiler will use the more specialized variant if the arguments so suggest.

So, assume an overloaded version of `sumVector()` is defined having the following prototype and a snippet of code requiring the instantiation of `sumVector`:

```
template <typename T>
T sumVector(T, unsigned);

extern int
    iArray[];

void fun()
{
    sumVector(iArray, 12);
}
```

The above example doesn't produce an ambiguity, even though the original `sumVector()` given in section 18.1.1 and the version declared here could both be used for the call. Why is there no ambiguity here?

In situations like this there is no ambiguity if both declarations are identical but for the fact that one version is able to accept a superset of the possible arguments that are acceptable for the other version. The original `sumVector()` template can accept only a pointer type as its first argument. The version declared here can accept a pointer type as well as any non-pointer type. A pointer type `iArray` is passed, so both template functions are candidates for instantiation. However, the original `sumVector()` template function can *only* accept a pointer type as its first argument. It is therefore more specialized than the one given here, and it is therefore selected by the compiler. If, for some reason, this is *not* appropriate, then an explicit template argument can be used to overrule the selection made by the compiler. E.g.,

```
sumVector<int *>(iArray, 12);
```

### 18.1.7 Selecting an overloaded (template) function

The following steps determine the actual function that is called, given a set of (template or non-template) overloaded functions:

- First, a set of candidate functions is constructed. This set contains all functions that are visible at the point of the call, having the same name as the function that is called. For a template function to be considered here, depends on the actual arguments that are used. These arguments must be acceptable given the standard template argument deduction process described in section 18.1.3. For example, assume all of the following declarations were provided:

```
template <typename T, typename U>
bool differentSigns(T t, U u);

bool differentSigns(double i, double j);
bool differentSigns(bool i, bool j);
bool differentSigns(int (&i)[2]);
```

Each of these functions will be included in the set of candidate functions in the following code fragment, as all of the four functions have the same name as the function that is called:

```
#include "candidates.h"

int main()
{
    differentSigns(int(), double());
}
```

- Second, the set of *viable functions* is constructed. Viable functions are functions for which type conversions exist that can be applied to match the types of the parameters of the functions and the types of the actual arguments. This removes the last two function declarations from the initial set: the third function is removed as there is no standard conversion from `double` to `int`, and the fourth function is removed as there is a mismatch in the number of arguments between the called function and the declared function.
- Third, the remaining functions are ranked in order of preference, and the first one is going to be used. Let's see what this boils down to:
  - For the template function, the function `differentSign<int, double>` is instantiated. For this function the types of the two parameters and arguments for a pairwise exact match: score two points for the template function.

- For the function `bool differentSigns(double i, double j)` the type of the second parameter is exactly matches the type of the second argument, but a (standard) conversion `int -> double` is required for the first argument: score one point for this function.

Consequently, the template function is selected as the one to be used. As an exercise, feed the above four declarations and the function `fun()` to the compiler and wait for the linker errors: ignoring the undefined reference to `main()`, the linker will complain that the (template) function

```
bool differentSigns<int, double>(int, double)
```

is an undefined reference.

If the template would have been declared as

```
template <typename T>
bool differentSigns(T t, T u);
```

then no template function would have been instantiated here. This is ok, as the ordinary function `differentSigns(double, double)` will now be used. An error occurs only if no instantiation of the template function can be generated and if no acceptable ordinary function is available. If such a case, the compiler will generate an error like

```
no matching function for call to 'differentSigns (int &, double &)
```

As we've seen, a template function in which all type parameters exactly match the types of the arguments prevails over an ordinary function in which a (standard) type conversion is required. Correspondingly, a template explicitly specialized function will prevail over an instantiation of the general template if both instantiations show an exact match between the types of the parameters and the arguments. For example, if the following template declarations are available:

```
template <typename T, typename U>
bool differentSigns(T t, U u);

template <> bool differentSigns<double, int>(double, int);
```

then the template explicitly specialized function will be selected without generating an extra instantiation from the general template definition.

Another situation in which an apparent ambiguity arises is when both an ordinary function is available and a proper instantiation of a template can be generated, both exactly matching the types of the arguments of the called function. In this case the compiler does not flag an ambiguity as the ordinary function is considered the more specialized function, which is therefore selected.

As a rule of thumb consider that when there are multiple viable functions sharing the top ranks of the set of viable functions, then the function template instantiations are removed from the set. If only one function remains, it is selected. Otherwise, the call is ambiguous.

### 18.1.8 Name resolution within template functions

Consider once more our function `sumVector()` of section 18.1.1, but now it's given a somewhat different implementation:

```

template <typename T>
T sumVector(T *array, unsigned n)
{
    T
        sum = accumulate(array, array + n, T());

    cout << "The array has " << n << " elements." << endl;
    cout << "The sum is " << sum << endl;

    return sum;
}

```

In this template definition, `cout`'s `operator<<()` is called to display, a `char const *text`, an unsigned, and a `T`-value. The first `cout` statement displays the text and the unsigned value, no matter what happens in the template. These types *do not depend on a template parameter*. If a type does not depend on a template parameter, the necessary declarations for compiling the statement must be available when the definition of the template is given. In the above template definition this implies that

```
ostream &ostream::operator<<(unsigned)
```

and

```
ostream &ostream::operator<<(char const *)
```

must be known to the compiler when it reads the definition of the template. On the other hand,

```
cout << ... << sum << endl
```

cannot be compiled by the time the template's definition is given, as the type of the variable `sum` *depends on a template parameter*. The statement can therefore only be checked for semantical correctness (i.e., the question whether `sum` can actually be inserted into `cout`) using `operator<<()` at the point where the template function is instantiated.

Names (variables) whose type depend on a template parameter are resolved when the template is instantiated: at that point the relevant declarations must be available. The location where this happens is called the template's *point of instantiation*. As a rule of thumb, make sure that the necessary declarations (usually: header files) are available at every point of instantiation of the template.

## 18.2 Template classes

Like templates for functions, templates can be constructed for complete classes. The construction of a template class can be considered when the class should be available for different types of data. Template classes are frequently used in C++: chapter 12 covers general data structures like `vector`, `stack` and `queue`, which are available as *template classes*. With template classes, the algorithms and the data on which the algorithms operate are completely separated from each other. To use a particular data structure on a particular data type, only the data type needs to be specified at the definition or declaration of the template class object, e.g., `stack<int> istack`.

In the upcoming sections the construction of template classes is discussed. In a sense, template classes compete with object oriented programming (cf. chapter 14), where a similar mechanism is seen. Polymorphism allows the programmer to separate algorithms from data, by deriving classes from the base class in which the algorithm is implemented, while implementing the data in the derived class, together with member functions that were defined as pure virtual functions in the base class to handle the data.

Generally, template classes are easier to use. It is certainly easier to write `stack<int> istack` to create a stack of ints than it is to derive a new class `Istack`: `public stack` and to implement all necessary member functions to be able to create a similar stack of ints using object oriented programming. On the other hand, for each different type that is used with a template class the complete class is reinstantiated, whereas in the context of object oriented programming the derived classes *use*, rather than *copy*, the functions that are already available in the base class.

Below a simple version of the template class `Vector` is constructed: the essential characteristics of a template class are illustrated, without attempting to redo the existing `vector` class completely.

### 18.2.1 Template class definitions

The construction and use of template classes will be covered in the next sections, in which a simple template class `Vector` will be constructed.

The construction of a template class can normally begin with the construction of a normal class interface around a hypothetical type `Type`. If more hypothetical types are required, then hypothetical types `U`, `V`, `W`, etc. can be used as well. Assume we want to construct a class `Vector`, that can be used to store values of type `Type`. We want to provide the class with the following members:

- Constructors to create an object of the class `Vector`, possibly of a given size, as well as a copy constructor, since memory will be allocated by the object to store values of type `Type`.
- A destructor.
- An overloaded `operator=()`.
- An `operator[]()` to retrieve and reassign the elements given their indices.
- Forward and backward iterators to be able to visit all elements sequentially, either from the first to the last or from the last to the first.
- A member `push_back()` to add a new element at the end of the vector.

Should the set of members include members that can be used with `const` objects? In practical situations it probably should, but for now these members are not included in the interface: We've left them for the reader to implement.

Now that we have decided which members we want, the class interface can be constructed. Like template functions, a template class definition begins with the keyword `template`, which is followed by a non-empty list of template type and/or non-type parameters, surrounded by angular brackets. This template announcement is followed by the class interface, in which the template parameters may be used to represent types and constants. Here is the first form of the class interface of the `Vector` template class, already containing the private member function `construct`, which is used in the implementation of the copy constructor, the destructor, and the overloaded assignment operator. The class also already contains an iterator type: it's simply defined as a pointer to an element of the vector. The `reverse_iterator` will be added later. Note that the `Vector` template class contains only one template type parameter, and no non-type parameter.



```

template <typename Type>
class Vector
{
    public:
        typedef reverse_iter<Type> reverse_iterator;

        Vector();
        Vector(unsigned n);
        Vector(Vector<Type> const &other);
        ~Vector();
        Vector<Type> const &operator=(Vector<Type> const &other);
        Type &operator[](int index);
        Vector<Type> &sort();
        void push_back(Type const &value);
        Type *begin();
        Type *end();
        reverse_iterator rbegin();
        reverse_iterator rend();
        unsigned size();
    private:
        void construct(Vector<Type> const &other);
        Type
            *start,
            *finish,
            *end_of_storage;
};

```

Within the class interface definition the abstract type `Type` can be used as a normal typename. However, note the `Vector<Type>` constructions appearing in the interface: there is no plain `Vector`, as the `Vector` will be bound to a type `Type`, to be specified later on in the program using a `Vector`.

Different from template functions, template class parameters can have default arguments. This holds true both for template type- and template non-type parameters. If a template class is instantiated without specifying arguments for the template parameters, and if default template parameters were defined, then the defaults are used. Such defaults should be suitable for a majority of instantiations of the class. E.g., for the template class `Vector` the template announcement could have been altered to specify `int` as the default type:

```

template <typename Type = int>

```

The class contains three data members: pointers to the begin and end of the allocated storage area (respectively `data` and `end_of_storage`) and a pointer pointing just beyond the element that was last allocated. The allocation scheme will add extra elements beyond the ones that are actually required to reduce the number of times the vector must be reallocated to accomodate new elements.

Template classes may be *declared* by removing the template interface definition (the part between the curly braces), replacing the definition by a semicolon:

```

template <typename Type>
class Vector;

```

here too, default types may be specified.

### 18.2.2 Template class instantiations

template classes are instantiated when an object of a template class is defined. When a template class object is defined (or declared) the template parameters must explicitly be specified (note that the parameters having default arguments are also specified, albeit as defaults). Template arguments are never deducted, as with template functions. To define a `Vector` to store ints, the construction

```
Vector<int>
    vInt;
```

is used. For a `Vector` for strings

```
Vector<string>
    vString;
```

is used.

In combination with the keyword `extern` template class objects can be *declared* rather than *defined*. E.g.,

```
extern Vector<int>
    vInt;
```

Template (type) parameters can be used to specify a template type of a template that is used within a template (for example, when a template class uses composition involving an object of another template class). In the following example a template function `cloneVector()` is defined, using type parameter `T`. It receives, defines, and returns `Vector` references and objects:

```
template <typename T>
Vector<T> cloneVector(Vector<T> &vector)
{
    return Vector<T>(vector);
}
```

A template class is not instantiated if a reference or pointer to the class template is used. In the above example, the `return Vector<int>(vector)` statement results in a template instantiation, but the parameter of the function, being a reference, won't result in a template instantiation. However, if a member function of a template class is used with a pointer or reference to a template class object, then the class *is* instantiated. E.g., in the following code

```
#include "vector.h"

void add(Vector<int> &vector, int value)
{
    vector.push_back(value);
}
```

the class `Vector<int>` will be instantiated.

### 18.2.3 Non-type parameters

Template nontype parameters must be constant expressions. I.e., the compiler must be able to evaluate their values. For example, the following class uses a template type parameter to define the type of the elements of a buffer, and a template nontype parameter to define the size of the buffer:

```
template <typename Type, unsigned size>
class Buffer
{
    public:
        Type
        buffer[size];
};
```

The `size` parameter must be a constant value when a `Buffer` object is defined or declared. E.g.,

```
Buffer<int, 20>
    buffer;
```

Note that

- Global variables have constant addresses, that 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 for the specification of 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 an unsigned parameter is specified, an int may be used too.

### 18.2.4 Template class member functions

Normal design considerations should be followed when constructing template class member functions or template class constructors: template class type parameters should preferably be defined as `T const &`, rather than `T`, 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, again to prevent double assignment of composed objects: once by the default constructor of the object, once by the assignment itself.

Template member functions must be known to the compiler when the template is instantiated. The current Gnu g++ compiler does not support precompiled template classes, therefore the member

functions of templates are inline functions. They can be defined inside the template interface or outside the template interface. Inline template member functions are defined in the same way as inline member functions of non-templates classes. However, for the member functions that are defined outside of the template's interface

- No *inline* keyword is required in the interface,
- A template `<template parameter list>` definition is required.

In the Vector template class a member function

```
Type &operator[](unsigned index) throw(char const *)
```

could be declared. When defined outside of the template's interface, its implementation would be:

```
template <typename T>
T &Vector<T>::operator[](unsigned index) throw(char const *)
{
    if (index >= (beyond - data))
        throw "Vector array index out of bounds";
    return data[index];
}
```

Note the fact that the class type of `operator[]()` is the generic `Vector<T>` type. The abstract data type `T` is used to define the type of the return value of the member function.

### 18.2.5 Template classes and friend declarations

Template classes may declare other functions and classes as friends. There are three types of friend declarations that can appear within a template class:

- A *nontemplate friend function* or class. This is a well-known friend declaration, such as the insertion operator for ostream objects.
- A *bound friend template* class or function. Here the template parameters of the current template are used to *bind* the types of another template class or function, so that a one-to-one correspondence between the template's parameters and the template parameters of the friend template class or function is obtained.
- A *unbound friend template* class or function. Here the template parameters of the friend template class or function remain to be specified, and are not related in some predefined way to the current template's parameters.

In the following sections the three types of friend declarations will be discussed in some detail.

#### Non-template friends

A template class may declare another function or class or class member function as its friend. Such a friend may access the private members of the template class. Classes and ordinary functions can be declared as friend, but a class interface must have been seen by the compiler before one of its

members can be declared a friend of a template class (in order to verify the name of the friend function against the interface).

For example, here are some friend declarations:

```
class Friend
{
    public:
        void member();
};

template <typename T>
class Vector
{
    friend class AnotherFriend;    // declaration only is ok here
    friend void anotherMember();   // declaration is ok here
    friend Friend::member();       // Friend interface class required.
};
```

Such ordinary friends can be used, e.g., to access static private members of a class or they can themselves define objects of the class declaring them as friends to access all members of these objects.

### Bound friends

With *bound friend* template classes or functions there is a one-to-one mapping between the types that are used with the instantiations of the friends and the template class declaring them as friends. Here the friends are themselves templates. For example:

```
template <typename T>
class Friend;           // declare a template class

template <typename T>
void function(Friend<T> &t); // declare a template function

template <typename T>
class AnotherFriend
{
    public:
        void member();    // a member function within a class
}

template <typename T>
class Vector
{
    friend class Friend<T>;           // 1 (see the text)
    friend void function<T>(Friend<T> t); // 2
    friend void AnotherFriend<T>::member(); // 3
};
```

Above, three friend declarations are illustrated:

- At 1, the class `Friend` is declared a friend of `Vector` if it is instantiated for the same type `T` as `Vector` itself.
- At 2, the function `function` is declared a friend of `Vector` if it is instantiated for the same type `T` as `Vector` itself. Note that the template type parameter `T` appears immediately following the function name in the friend declaration. Here the correspondence between the function's template parameter and `Vector`'s template parameter is defined. After all, `function()` could have been a parameterless function. Without the `<T>` affixed to the function name, it is an ordinary function, expecting an (unrestricted) instantiation of the class `Vector` for its argument.
- At 3, a specific member function of the class `AnotherFriend`, instantiated for type `T` is declared as a friend of the class `Vector`.

Assume we would like to be able to insert the elements of a `Vector` into an `ostream` object, using `operator<<()`. For such a situation the `copy()` generic algorithm in combination with the `ostream_iterator<>()` comes in handy. However, the latter iterator is a template function, depending on type `T`. If we can assume that `data` and `beyond` can be used as iterators of `Vector`, then the implementation is quickly realized by defining `operator<<()` as a bound template function, and by declaring this operator as a friend of the class `Vector()`:

```
#include <algorithm>
#include <iostream>

template<typename T>
class Vector
{
    friend ostream &operator<< <T> (ostream &str,
                                   Vector<T> const &vector);

private:
    T
        *data,
        *beyond;
};

template <typename T>
ostream &operator<<(ostream &str, Vector<T> const &vector)
{
    copy(vector.data, vector.beyond, ostream_iterator<T *>(str, " "));
    return str;
}
```

## Unbound friends

By prepending the `friend` keyword to the `template<typelist>` phrase, friends received their own template parameter list. The template types of these friends are completely independent from the type of the template class declaring the friends. Such friends are called *unbound friends*. Every instantiation of an unbound friend has unrestricted access to the private members of every instantiation of the template class declaring the friends.

Here is the syntactic convention for declaring an unbound friend function, an unbound friend class and an unbound friend member function of a class:

```
template <typename Type>
```

```

class Vector
{
    template <typename T>
    friend void function(); // unbound friend function

    template <typename T>
    friend class Friend;    // unbound friend class

    template <typename T>    // unbound friend member function
    friend void AnotherFriend<T>::member();
};

```

### 18.2.6 Template classes and static data

When static members are defined in a template class, these static members are instantiated for every different instantiation of the template class. As they are static members, there will be only one member when multiple objects of the same template type(s) are defined. For example, in a class like:

```

template <typename Type>
class TheClass
{
    private:
        static int
            objectCounter;
};

```

There will be *one* `TheClass<Type>::objectCounter` for each different `Type`. However, the following will result in just one static variable, which is shared among the different objects:

```

TheClass<int>
    theClassOne,
    theClassTwo;

```

Remember that static members are only *declared* in their classes. They must be *defined* separately. With static members of template classes this is not different. But, comparable to the implementations of static functions, the definitions of static members are usually provided in the same file as the template class interface itself. The definition of the static member `objectCounter` is therefore:

```

template <typename Type>
class TheClass
{
    private:
        static int                // declaration
            objectCounter;
};

template <typename Type>          // definition
int
    TheClass<Type>::objectCounter = 0;

```

In the above case `objectCounter` is an `int`, and thus independent of the template type parameter `Type`. In a list-like construction, where a pointer to objects of the class itself is required, the template type parameter `Type` does enter the definition of the static variable, as is shown in the following example:

```
template <typename Type>
class TheClass
{
    private:
        static TheClass
            *objectPtr;
};

template <typename Type>
TheClass<Type>
    *TheClass<Type>::objectPtr = 0;
```

Note here that the definition can be read, as usual, from the variable name back to the beginning of the definition: `objectPtr` of the class `TheClass<Type>` is a pointer to an object of `TheClass<Type>`.

### 18.2.7 Derived Template Classes

Template classes can be used in class derivation as well. Consider the following base class:

```
template<typename T>
class Base
{
    public:
        Base(T const &t)
        :
            t(t)
        {}
    private:
        T const
            &t;
};
```

The above class is a template class, which can be used as a base class for the following derived template class `Derived`:

```
template<typename T>
class Derived: public Base<T>
{
    public:
        Derived(T const &t)
        :
            Base(t)
        {}
};
```

Other combinations are possible as well: By specifying concrete template type parameters of the base class, the base class is instantiated and the derived class becomes an ordinary (non-template)



class:

```
class Ordinary: public Base<int>
{
    public:
        Ordinary(int x)
        :
            Base(x)
        {}
};

// With the following object definition:
Ordinary
    o(5);
```

This construction allows us in a specific situation to add functionality to a template class, without the need for constructing a derived template class.

## 18.2.8 Nesting and template classes

When a class is nested within a template class, it automatically becomes a template class itself. The nested class may use the template parameters of the surrounding class, as shown in the following skeleton program. Within the class `Vector` the class `reverse_iterator` is defined. The class receives its information from the surrounding class, a `Vector<Type>` class. Since this surrounding class is the only class which should be able to construct its reverse iterators, the constructor of `reverse_iterator` is made private, and the surrounding class is given access to the private members of `reverse_iterator` using a *bound friend* declaration.

Also, the surrounding class is allowing access to its private members by the class `reverse_iterator`, so once the definition of the class `reverse_iterator` has been seen by the compiler, we're ready to declare `reverse_iterator` a friend of `Vector<Type>`.

Once the classes are set up, a main program constructs a `Vector<int>`, which is used to construct a `Vector<int>::reverse_iterator`, using its `rbegin()` member. This member, however, returns an anonymous (and thus: constant) `reverse_iterator` object. So the public copy constructor is used to make a non-constant `reverse_iterator` object available in `main()`, as the `rbegin` object. Here is the skeleton program, omitting all data members and extra member functions:

```
#include <iostream>

template <typename Type>
class Vector
{
    class reverse_iterator
    {
        friend class Vector<Type>;

    public:
        reverse_iterator(reverse_iterator const &other);
        Type &operator*() const;
    private:
        reverse_iterator(Vector<Type> *vector);
    };
};
```

```

        friend reverse_iterator;

    public:
        Vector();
        reverse_iterator rbegin()
        {
            return reverse_iterator(this);
        }
};

int main()
{
    Vector<int>
        vi;

    Vector<int>::reverse_iterator
        rbegin = vi.rbegin();

    cout << *rbegin;
}

```

Nested enumerations and typedefs can be defined in template classes as well. For example, with arrays the distinction between the last index that can be used and the number of elements frequently causes confusion in people who are exposed to C-array types for the first time. The following construction automatically provides valid `last` and `nElements` definition:

```

template<typename Type, int size>
class Buffer
{
    public:
        enum Limits
        {
            last = size - 1,
            nElements
        };
        typedef Type elementType;

        Buffer()
        :
            b(new Type [size])
        {}
    private:
        Type
            *b;
};

```

This small example defines `Buffer<Type, size>::elementType`, `Buffer<Type, size>::last` and `Buffer<Type, size>::nElements` (as values), as well as `Buffer<Type, size>::Limits` and `Buffer<Type, size>::elementType` (as typenames).

Of course, the above represents the template form of these values and declarations. They must be instantiated before they can be used. E.g,

Buffer<int, 80>::elementType

is a synonym of int.

Note that a construction like Buffer::elementType is illegal, as the type of the Buffer class remains unknown.

### 18.2.9 Member templates

Template classes or functions can also be defined within other classes (which itself may or may not be a template class). Such a nested template function or class is called a *member template*. It is defined the same way as any other ordinary template class, including the template <typename ...> header. E.g.,

```
template <typename Type1>
class Outer
{
    public:
        template <typename Type2>           // template class
        class Inner
        {
            public:
                Type1
                variable1;
                Type2
                variable2;
        };

        template <typename Type3>           // template function
        Type3 process(Type3 const &p1, Type3 const &p2)
        {
            Type3
            result;

            return result;
        }
};
```

The special characteristic of a member template is that it can use its own and its surrounding class' template parameters, as illustrated by the definition of variable1 in the class Inner. Here are some consequences of using templates in classes

- Normal access rules apply: the function process() can be used by the general program, given an instantiated Outer object. Of course, this implies that a large number of possible instantiations of process() are possible. Actually, an instantiation is only then constructed when a process() function is in fact used. In the following code the member function int process(int const &p1, int const &p2) is instantiated, even though the object is of the class Outer<double>:

```
Outer<double>
outer;

outer.process(10, -3);
```

The `process()` template member function allows the processing of any other type by an object of the class `Outer`.

- Any function can be defined as a template function, not just an ordinary member function. A constructor can be defined as a template member as well:

```
template <typename Type1>
class Outer
{
    public:
        template <typename Type2>    // template class
        Outer(Type2 const &initialValue)
        {
        }
};
```

Here, an `Outer` object can be constructed for a particular type given another type that's given to the constructor. E.g.

```
Outer<int>
    t(12.5);    // instantiates Outer(double const &initialvalue)
```

- Template members can be defined inline or outside of their containing class. When a member is defined outside of its surrounding class, the template parameter list must precede the template parameter list of the template member. E.g.,

```
template <typename Type1>
class Outer
{
    public:
        template <typename Type2>    // template class
        class Inner;

        template <typename Type>    // template function
        Type process(Type const &p1, Type const &p2);
};

template <typename Type1> template <typename Type2>    // template class member
class Outer<Type1>::Inner    // no Type2 with 'Inner'
{
    public:
        Type1
            variable1;
        Type2
            variable2;
};

template <typename Type1> template <typename Type>    // template function member
Type Outer<Type1>::process(Type const &p1, Type const &p2)
{
    Type
        result;
    return (result);
}
```

Note the way the class `Inner` starts its implementation: with the `Outer` class template parameters are used, with the `Inner` class definition the template parameters must be omitted.

- When template member classes are not defined inline, the formal template parameter names in the declaration and implementation must be identical.

### 18.2.10 Template class specializations

Template class specializations are used in cases where template member functions cannot be used with a (class) type for which the template is instantiated. In those cases template member functions can be explicitly constructed to suit the needs of the particular type for which the template is instantiated. These explicitly constructed members are called template class specializations.

Assume we have a template class which supports the insertion of its type parameter into an `ostream`. E.g.,

```
template <typename Type>
class Inserter
{
    public:
        Inserter(Type const &t)
        :
            object(t)
        {}
        ostream &insert(ostream &os) const
        {
            return os << object;
        }
    private:
        Type
            object;
};
```

In the example a plain member function is used to insert the current object into an `ostream`. The implementation of the `insert()` function shows that it uses the `operator<<`, as defined for the type that was used when the template class was instantiated. E.g., the following little program instantiates the class `Inserter<int>`:

```
int main()
{
    Inserter<int>
        ins(5);

    ins.insert(cout) << endl;
}
```

Now suppose we have a class `Person` having the following member function:

```
class Person
{
    public:
        ostream &insert(ostream &ostr) const;
};
```

This class cannot be used to instantiate `Insertter`, as it does not have a `operator<<()` function, which is used by the function `Insertter<Type>::insert()`. Attempts to instantiate `Insertter<Person>` will result in a compilation error. For example, consider the following `main()` function:

```
int main()
{
    Person
        person;
    Insertter<Person>
        p2(person);

    p2.insert(cout) << endl;
}
```

If this function is compiled, the compiler will complain about the missing function `ostream & << const Person &`, which was indeed not available. However, the function `ostream &Person::insert(ostream &ostr)` is available, and it serves the same purpose as the required function `ostream & Insertter<Person>::insert(ostream &)`.

For this situation multiple solutions exist. One would be to define an `operator<<(Person const &p)` function which calls the `Person::insert()` function. But in the context of the `Insertter` class, this might not what we want. Instead, we might want to look for a solution that is closer to the class `Insertter`.

Such a solution exists in the form of a *template class specialization*. Such an *explicit specialization definition* starts with the word `template`, then two angular brackets (`<>`), which is then followed by the function definition for the instantiation of the template class for the specific template parameters. So, with the above function this yields the following function definition:

```
template<>
ostream &Insertter<Person>::insert(ostream &os) const
{
    return object.insert(os);
}
```

Here we explicitly define a function `insert` of the class `Insertter<Person>`, which calls the appropriate function that lives in the `Person` class.

Note that the explicit specialization definition is a true *definition*: it should *not* be given in the header file of the `Insertter` template class, but it should have its own source file. However, in order to inform the compiler that an explicit specialization is available, it can be *declared* in the template's header file. The declaration is straightforward: the code-block is replaced by the semicolon:

```
template<>
ostream &Insertter<Person>::insert(ostream &os) const;
```

It is also possible to specialize a complete template class. The following shows such a specialization for the above class `Insertter` applied to the `double` primitive type:

```
template <>
class Insertter
{
    public:
```

```

        Inserter<double>(double const &t);
        ostream &insert(ostream &os) const;
    private:
        double
            object;
};

```

The explicit template class specialization is obtained by replacing all references to the template's class name `Inserter` by the class name and the type for which the specialization holds true: `Inserter<double>`, and by replacing occurrences of the template's type parameter by the actual type for which the specialization was constructed. The template class specialization interface must be provided *after* the interface of the original template class. The definition of its members are, analogously to the `Inserter<Person>::insert()` function above, given in separate source files. However, in the case of a complete template class specialization, the definitions of its members should *not* be preceded by the `template<>` prefix. E.g.,

```

Inserter<double>(double const &t)    // NO template<> prefix !
:
    object(t)
{}

```

### 18.2.11 Template class partial specializations

In cases where a template has more than one parameter, a *partial specialization* rather than a full specialization might be appropriate. With a partial specialization, a subset of the parameters of the original template can be redefined.

Let's assume we are working on a image processing program. A class defining an image receives two int template parameters, e.g.,

```

template <int columns, int rows>
class Image
{
    public:
        Image()
        {
            // uses 'columns' and 'rows'
        }
};

```

Now, assume that an image having 320 columns deserves special attention, as those pictures allow us to use, e.g., a special smoothing algorithm. From the general template given above we can now construct a *partially specialized* template, which *only* has a `columns` parameter. Such a template is like an ordinary template parameter, in which only the `rows` remain as a template parameter. When the specialized class is defined, the specialization is made explicit by mentioning a specialization parameter list:

```

template <int rows>
class Image<320, rows>
{
    public:
        Image()

```

```

    {
        // use 320 columns and 'rows' rows.
    }
};

```

With the above partially specialized template definition the 320 columns are explicitly mentioned at the class interface, while the rows remain variable. Now, if an image is defined as

```

Image<320, 240>
    image;

```

two instantiations *could* be used: the fully general template is a candidate as well as the partially specialized template. Since the partially specialized template is *more specialized* than the fully general template, the `Image<320, rows>` template will be used. This is a general rule: a more specialized template instantiation is chosen in favor of a more general one wherever possible.

Every template parameter can be used for the specialization. In the last example the columns were specialized, but the rows could have been specialized as well. The following partial specialization of the template class `Image` specializes the rows parameter and leaves the columns open for later specification:

```

template <int columns>
class Image<columns, 200>
{
    public:
        Image()
        {
            // use 'columns' columns and 200 rows.
        }
};

```

Even when *both* specializations are provided there will (generally) be no problem. The following three images will result in, respectively, an instantiation of the general template, of the template that has been specialized for 320 columns, and of the template that has been specialized for the 200 rows:

```

Image<1024, 768>
    generic;
Image<320, 240>
    columnSpecialization;
Image<480, 200>
    rowSpecialization;

```

With the generic image, no specialized template is available, so the general template is used. With the `columnSpecialization` image, 320 columns were specified. For that number of columns a specialized template is available, so it's used. With the `rowSpecialization` image, 200 rows were specified. For that number of rows a specialized template is available, so that specialized template is used with `rowSpecialization`.

One might wonder what happens if we want to construct the following image:

```

Image<320, 200>
    superSpecialized;

```



Is this a specialization of the columns or of the rows? The answer is: neither. It's ambiguous, precisely because *both* the columns *and* the rows could be used with a (differently) specialized template. If such an image is required, yet another specialized template is needed, albeit that that one is not a *partially* specialized template anymore. Instead, it specializes all its parameters with the class interface:

```
template <>
class Image<320, 200>
{
    public:
        Image()
        {
            // use 320 columns and 200 rows.
        }
};
```

The above super specialization of the Image template will be used with the image having 320 columns and 200 rows.

### 18.2.12 Name resolution within template classes

In section 18.1.8 the name resolution process with template functions was discussed. As is the case with template functions, name resolution in template classes also proceeds in two steps. Names that do not depend on template parameters are resolved when the template is defined. E.g., if a member function in a template class uses a `qsort()` function, then `qsort()` does not depend on a template parameter. Consequently, `qsort()` must be known when the compiler sees the template definition, e.g., by including the file `cstdlib` or `stdlib.h`.

On the other hand, if a template defines a `<typename Type>` template parameter, which is the returntype of some template function, e.g.,

```
Type returnValue();
```

then we have a different situation. At the point where template objects are defined or declared; at the point where template member functions are used; and at the point where static data members of template classes are defined or declared, it must be able to resolve the template type parameters. So, if the following template class is defined:

```
template <typename Type>
class Resolver
{
    public:
        Resolver();
        Type result();
    private:
        Type
            datum;
        static int
            value;
};
```

Then `string` must be known before each of the following examples can be compiled:

```

// ----- example 1: define the static variable
int Resolver<string>::value = 12;

// ----- example 2: define a Resolver object
int main()
{
    Resolver<string>
        resolver;
}

// ----- example 3: declare a Resolver object
extern Resolver<string>
    resolver;

```

## 18.3 Constructing iterators

In section 17.2 the iterators used with generic algorithms were introduced. We've seen that several types of iterators are distinguished by the generic algorithms: `InputIterators`, `ForwardIterators`, `OutputIterators`, `BidirectionalIterators` and `RandomAccessIterators`.

Also, in section 17.2 the characteristics of iterators were introduced: basically it should be possible to increment iterators, to dereference iterators and to compare iterators for equality.

However, in order to use iterators in the context of the generic algorithms iterators often must meet extra requirements. This is caused by the fact that generic algorithms check the types of the iterators they receive. Simple pointers are usually accepted, but if a class is used as an iterator, the iterator-class must be able to specify the kind of iterator it represents.

To make sure that an object of a class is interpreted as a particular type of iterator, the class must be derived from the classes `input_iterator`, `output_iterator`, `forward_iterator`, `bidirectional_iterator`, or `random_access_iterator`. These classes mainly define tags which are used by generic algorithms to check the correct types of the provided iterators, and have nothing to do with the actual implementation of the iterators. In order to derive classes from the iterator-classes, sources must

```
#include <iterator>
```

To construct a class `Iterator` that can be used as an iterator (which could be an ordinary iterator, a `const_iterator`, a `reverse_iterator` or a `const_reverse_iterator`) for type 'Type' (which may be a template type parameter or a specific type like an `int` or a `string`), one must be prepared to implement several members. Also, these iterator-classes must be derived from appropriate iterator-tag classes. With the exception of the `output_iterator` base class, these iterator classes are template classes themselves, using `Type` and `ptrdiff_t` as their template class parameters. The following iterator classes can thus be constructed:

- An `InputIterator`:
  - Base class: `input_iterator<Type, ptrdiff_t>`
  - Type `const &operator*() const`: the dereference operator.
  - `Iterator &operator++()`: the prefix increment operator.
  - `bool operator==(Iterator const &other)`: the operator to compare two iterators for equality.

- An OutputIterator:
  - Base class: `output_iterator` (Note: no template class)
  - `Type &operator*() const`: the dereference operator.
  - `Iterator &operator++()`: the prefix increment operator.
- A ForwardIterator:
  - Base class: `forward_iterator<Type, ptrdiff_t>`
  - `Type &operator*() const`: the dereference operator.
  - `Iterator &operator++()`: the prefix increment operator.
- A BidirectionalIterator:
  - Base class: `bidirectional_iterator<Type, ptrdiff_t>`
  - The same members as required for the ForwardIterator
  - `:`: the postfix increment operator.
  - `Iterator &operator--()`: the prefix decrement operator.
  - `Iterator &operator--(int)`: the postfix decrement operator.
- A RandomAccessIterator:
  - Base class: `random_access_iterator<Type, ptrdiff_t>`
  - The same members as required for the BidirectionalIterator
  - `Iterator operator+(int step)`: the random step forward operator.
  - `Iterator operator-(int step)`: the random step backward operator.
  - `:`: the pointer arithmetic operator.
  - `:`: the comparison operator applied to the Iterators themselves. Note that this is *not* a comparison applied to two `Type` values: with the `RandomAccessIterator` it must be possible to order the iterators themselves, so they can be compared with respect to their relative magnitudes.

Section 19.6 shows the implementation of the most complex of these iterators, the `RandomAccessIterator`, in the context of the `Vector` class.

## Chapter 19

# Concrete examples of C++

This chapter presents a number of concrete examples of programming in C++. Items from this document such as virtual functions, static members, etc. are rediscussed. Examples of container classes are shown.

Also, with the advent of the ANSI/ISO standard, classes supporting streams based on *file descriptors* are no longer available, including the Gnu `procbuf` extension. These classes were frequently used in older C++ programs. This section of the C++ Annotations develops an alternative: classes extending `streambuf`, allowing the use of file descriptors, and classes around the `fork()` system call.

Another example digs into the peculiarities of using a parser- and scanner-generator with C++. Once the input for a program exceeds a certain level of complexity, it's advantageous to use a scanner- and parser-generator for creating the code which does the actual input recognition. The example describes the usage of these tool in a C++ environment.

### 19.1 'streambuf' classes using file descriptors

#### 19.1.1 A class for output operations

Usually extensions to the ANSI/ISO standard are available allowing us to read from and/or write to *file descriptors*. However, these extensions are not standard, and may thus vary across compilers. As a file descriptor can be considered a device, it seems natural to use the class `streambuf` as the starting point for the construction of classes interfacing file descriptors. In this section we will discuss the construction of a class which may be used to write to a device defined by its file descriptor: it may be a file, but could be a pipe or socket. Section 19.1.2 will discuss the matter of reading from devices that are known by their file descriptors, while section 19.3 will reconsider the question of redirection, which was discussed earlier in section 5.8.3.

Basically, deriving a class for output operations is very simple. The only member function that must be overridden is the virtual `int overflow(int c)` member. This member is responsible for writing characters to the device in the case where there is no buffer or no more room in the buffer. If `fd` represents the file descriptor to write to, and if we decide against using a buffer, the member `overflow()` can be as simple as:

```
class UnbufferedFD: public std::streambuf
{
```

```

public:
    int
    overflow(int c)
    {
        if (c != EOF)
        {
            if (write(fd, &static_cast<char>(c), 1) != 1)
                return EOF;
        }
        return c;
    }
    ...
}

```

The received argument is either written as char to the file descriptor, or EOF is returned. However, this approach does not use an output buffer. As the use of a buffer is strongly advised (see also the next section), we will discuss the construction of a class using an output buffer in somewhat greater detail.

When an output buffer is to be used, the `overflow()` member gets a bit more complex, due to the fact that it is now called only when the buffer is full. So, *first* we will have to flush the buffer, for which the (virtual) function `streambuf::sync()` should be used. Defining `sync()` and using it in `overflow()` is not all that must be done: the information to write might be less than the size of the buffer. So, at the end of the lifetime of our special `streambuf` object, it might find itself having a buffer that is only partially full. So, we must make sure that the buffer is flushed by the time our object goes out of scope. This is of course very simple: `sync()` should be called in the destructor as well.

Now that we've considered the consequences of using an output buffer, we're almost ready to construct our derived class. We add a couple of extra features. First, we permit the specification of the size of the output buffer by the user of the class. Second, we permit the construction of an object of the class before the file descriptor to use is actually known. Later, in section 19.4 we'll encounter a situation where this feature will be used. In order to save some space, the successful operation of various calls were not checked. In 'real life' implementations these checks should of course not be omitted. Here is the class `ofdnstreambuf`:

```

#ifndef _OFDNSTREAMBUF_H_
#define _OFDNSTREAMBUF_H_

#include <unistd.h>
#include <streambuf>
#include <ios>

class ofdnstreambuf: public std::streambuf
{
public:
    ofdnstreambuf()
    :
        bufsize(0),
        buffer(0)
    {}

    ofdnstreambuf(int fd, unsigned bufsize = 1)
    {
        open(fd, bufsize);
    }
}

```

```

    }

    ~ofdstreambuf()
    {
        if (buffer)
        {
            sync();                // 23
            delete buffer;
        }
    }

    void open(int xfd, unsigned xbufsize = 1)
    {
        fd = xfd;
        bufsize = xbufsize == 0 ? 1 : xbufsize;

        buffer = new char[bufsize];
        setp(buffer, buffer + bufsize);    // 34
    }

    int sync()
    {
        if (pptr() > pbase())
        {
            write(fd, buffer, pptr() - pbase());    // 41
            setp(buffer, buffer + bufsize);    // 42
        }
        return 0;
    }

    int overflow(int c)
    {
        sync();                // 49
        if (c != EOF)
        {
            *pptr() = static_cast<char>(c);    // 52
            pbump(1);
        }
        return c;
    }

private:
    unsigned bufsize;
    int fd;
    char *buffer;
};

#endif

```

- At line 23, in the destructor, `sync()` is called to write any unprocessed characters in the output buffer to the device.
- At line 34, in the `open()` member, the buffer is initialized. Using `setp()` the begin and end points of the buffer are passed to the `streambuf` base class, allowing it to setup `pbase()` `pptr()` and `epptr()`.

- At line 41 in the member `sync()`, the unflushed characters in the buffer are written to the device. Then, at the next line, the buffer is reinitialized. Note that `sync()` should return 0 after a successful flush operation.
- At line 42 the buffer pointers maintained by `streambuf` are reset to their initial values by calling `setp()` once again.
- At lines 49 and 52, in the member `overflow()`, `sync()` is called to flush the now filled up output buffer to the device. As this recreates an empty buffer, the character `c` which could not be written to the buffer is now entered into the buffer using the member functions `pptr()` and `pbump()`. Notice that entering a character into the buffer is realized using available member functions of `streambuf`.

Depending on the *number* of arguments, the following program uses the `ofdstreambuf` 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. Here is the program:

```
#include "fdout.h"
#include <string>
#include <iostream>
#include <istream>

using namespace std;

int main(int argc)
{
    ofdnstreambuf
        fds(STDOUT_FILENO, 500);
    ostream
        os(&fds);

    switch (argc)
    {
        case 1:
            os << "COPYING cin LINE BY LINE\n";
            for (string s; getline(cin, s); )
                os << s << endl;
            break;

        case 2:
            os << "COPYING cin BY EXTRACTING TO os.rdbuf()\n";

            cin >> os.rdbuf();      // Alternatively, use: cin >> &fds;
            break;

        case 3:
            os << "COPYING cin BY INSERTING cin.rdbuf() into os\n";
            os << cin.rdbuf();
            break;
    }
}
```

### 19.1.2 Classes for input operations

When a class performing input operations must be derived from the class `streambuf`, the class should use an input buffer of at least one character, to allow the use of the member functions `istream::putback()` or `istream::ungetc()`. Stream classes (like `istream`) normally allow the ungetting of at least one character using their member functions `putback()` or `ungetc()`. This is important, as these stream classes usually interface to `streambuf` objects. Although strictly speaking it is not necessary to implement a buffer in classes derived from `streambuf`, we strongly suggest to use buffers in these cases: the implementation is very simple and straightforward, and the applicability of such classes is greatly improved. Therefore, in all our classes derived from the class `streambuf` at least a buffer of one character will be defined.

#### Using a one-character buffer

When deriving a class (e.g., `ifdstreambuf`) from `streambuf` using a buffer of at least one character, at the minimum the member `streambuf::underflow()` should be overridden, as this is the member to which all requests for input eventually degenerate. Since also a buffer is needed, the member `streambuf::setg()` is used to inform the `streambuf` baseclass of the dimensions of the input buffer, so that it is able to set up its input buffer pointers so that `eback()`, `gptr()`, and `egptr()` return correct values. Here is the corresponding class definition:

```
#include <unistd.h>
#include <streambuf>
#include <ios>

class ifdstreambuf: public std::streambuf
{
public:
    ifdstreambuf(int fd)
    :
        fd(fd)
    {
        setg(buffer, buffer + 1, buffer + 1);    // 12
    }

    int
    underflow()
    {
        if (gptr() < egptr())                    // 18
            return *gptr();                      // 19

        if (read(fd, buffer, 1) <= 0)            // 21
            return EOF;

        setg(buffer, buffer, buffer + 1);        // 24
        return *gptr();

    }
protected:                                    // 27
    int    fd;
    char   buffer[1];
};
```



Please note the following:

- At line 12 the buffer is set up. In the private section a small array of one character is defined, and `setg()` will set `gptr()` equal to `egptr()`. Since this implies that the buffer is empty, `underflow()` will be called to refill the buffer.
- At lines 18 and 19 `underflow()` will first see whether the buffer is really empty. If not, then the character to be retrieved next is returned.
- At line 21 the buffer is refilled. If this fails (for whatever reason), EOF is returned. More sophisticated implementations could react more intelligently here, of course.
- At line 24 the buffer has been refilled, so `setg()` is called once again to set up `streambuf`'s buffer pointers correctly.
- Below line 27 two data members are defined: they were defined as protected data members so that derived classes (e.g., see section 19.1.2) can access them.

The above `ifdstreambuf` class is used in the following program:

```
#include "ifdbuf.h"
#include <iostream>
#include <istream>

using namespace std;

int main(int argc)
{
    ifdstreambuf
        fds(0);
    istream
        is(&fds);

    cout << is.rdbuf();
}
```

### Using an n-character buffer

How complex would things get if we decide to use a buffer of substantial size? Not that complex. The following class allows use to specify the size of a buffer, and it is basically the same class as `ifdstreambuf` from the previous section. To make things a bit more interesting, in the current class `ifdnstreambuf` the member `streambuf::xsgetn()` was also overridden, to optimize the reading of series of characters at once. Also, a default constructor is provided which can be used in combination with the added `open()` member in cases where we want to construct a `istream` object before we know the file descriptor that must be used. Later, in section 19.4 we'll encounter such a situation. In order to save some space, the successful operation of various calls were not checked. In 'real life' implementations these checks should of course not be omitted. Here is the new class `ifdnstreambuf`:

```
#ifndef _IFDNBUF_H_
#define _IFDNBUF_H_

#include <unistd.h>
```

```

#include <streambuf>
#include <ios>

class ifdnstreambuf: public std::streambuf
{
    public:
        ifdnstreambuf()
        :
            bufsize(0),
            buffer(0)
        {}

        ifdnstreambuf(int fd, unsigned bufsize = 1)
        {
            open(fd, bufsize);
        }

        ~ifdnstreambuf()
        {
            if (bufsize)
            {
                close(fd);
                delete buffer;                // 27
            }
        }

        void open(int xfd, unsigned xbufsize = 1)
        {
            fd = xfd;
            bufsize = xbufsize;
            buffer = new char[bufsize];
            setg(buffer, buffer + bufsize, buffer + bufsize);
        }

        int underflow()
        {
            if (gptr() < egptr())
                return *gptr();

            int nread = read(fd, buffer, bufsize);

            if (nread <= 0)
                return EOF;

            setg(buffer, buffer, buffer + nread);    // 49
            return *gptr();
        }

        std::streamsize
        xsgetn(char *dest, std::streamsize n)        // 54
        {
            int nread = 0;

            while (n)

```

```

    {
        if (!in_avail())
        {
            if (underflow() == EOF)          // 62
                break;
        }

        int avail = in_avail();

        if (avail > n)
            avail = n;

        memcpy(dest + nread, gptr(), avail); // 71
        gbump(avail);                        // 72

        nread += avail;
        n -= avail;
    }

    return nread;                            // 79
}

protected:
    int      fd;
    unsigned bufsize;
    char*     buffer;
};

#endif

```

With this class, please note:

- In line 27 the memory allocated for the buffer is deleted. Since the constructor dynamically allocates memory for the buffer, we must make sure that the allocated memory is returned to the common pool when the `ifdnstreambuf` object goes out of scope.
- At line 49 the buffer has just been refilled by `underflow()`. The input buffer pointers of `streambuf` are now reset in accord with the actual number of characters read from the device.
- At line 54 to 79 the function `xsgetn()` is overridden. In a loop `n` is reduced until 0, at which point the function terminates. Alternatively (line 62), the member returns if `underflow()` fails to obtain more characters. In lines 71 and 72 reading is optimized: instead of calling `streambuf::sbumpc()` `n` times, a block of `avail` characters is copied to the destination, using `streambuf::gpump()` to consume `avail` characters from the buffer using one function call.

The member function `xsgetn()` is called by `streambuf::sgetn()`, which is a `streambuf` member. The following example illustrates the use of this member function with a `ifdnstreambuf` object:

```

#include "ifdnbuf.h"
#include <iostream>
#include <istream>

using namespace std;

```

```

int main(int argc)
{
    ifdnstreambuf fds(0, 30); // internally: 30 char buffer
    char buf[80];             // main() reads blocks of 80
                               // chars

    while (true)
    {
        unsigned n = fds.sgetn(buf, 80);
        if (n == 0)
            break;
        cout.write(buf, n);
    }
}

```

### Seeking positions in 'streambuf' objects

When devices support *seek operations*, classes derived from `streambuf` should override `streambuf::seekoff()` and `streambuf::seekpos()`. In the following class `ifdseek`, which is derived from our previously constructed class `ifdstreambuf`, we assume that the device of which we have a file descriptor available supports these seeking operations. The class `ifdseek` is 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` will make sure that the input buffer is reset when a seek operation is requested. The class could also be derived from the class `ifdnstreambuf`, in which case the arguments to reset the input buffer must be adapted such that its second and third parameter point beyond the available input buffer. Here is the class definition:

```

#include "ifdbuf.h"
#include <unistd.h>

class ifdseek: public ifdstreambuf
{
    typedef std::streambuf::pos_type      pos_type;    // 6
    typedef std::streambuf::off_type      off_type;
    typedef std::ios::seekdir             seekdir;
    typedef std::ios::openmode            openmode;    // 9

public:
    ifdseek(int fd)                        // 12
    :
        ifdstreambuf(fd)
    {}

    pos_type                                     // 17
    seekoff(off_type offset, seekdir dir, openmode)
    {
        pos_type pos =
            lseek
            (
                fd, offset,
                (dir == std::ios::beg) ? SEEK_SET :
                (dir == std::ios::cur) ? SEEK_CUR :
                SEEK_END
            )
    }
}

```

```

        );

        if (pos < 0)
            return -1;

        setg(buffer, buffer + 1, buffer + 1);        // 32
        return pos;
    }

    pos_type                                          // 36
    seekpos(pos_type offset, openmode mode)
    {
        return seekoff(offset, std::ios::beg, mode);
    }
};

```

- In lines 6-9 several type names from the `streambuf` and `ios` classes were redefined to prevent us from specifying `std::streambuf` and `std::ios` again and again when referring to these types.
- At line 12 a plain constructor, deferring its job to its base class constructor is defined.
- At line 17 the function `lseek()` is used to seek a new position in a file whose file descriptor is known.
- At line 32, if the seeking succeeds, the `streambuf` class will reset its input buffer pointers to an empty buffer, so that `underflow()` will refill the buffer at the next request for input.
- At line 36 the companion function `seekpos` is overridden as well: it is actually defined by a call to `seekoff()`.

An example of a program using the class `ifdseek` is the following. If this program is given its own source file using input redirection then seeking is supported, and apart from the first line, every other line is shown twice:

```

#include "fdinseek.h"
#include <string>
#include <iostream>
#include <istream>
#include <iomanip.h>

using namespace std;

int main(int argc)
{
    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;
        }
    }
}

```

buffer is refilled

### Multiple 'unget()' calls in 'streambuf' objects

As stated earlier `streambuf` classes and classes derived from `streambuf` should at least support ungetting the character that was last read. Special care must be taken when series of `unget()` calls are to be supported. In this section we will show how to construct a class which is derived from `streambuf` and which allows a configurable number of `istream::unget()` or `istream::putback()` calls. Support for multiple (say 'n') `unget()` calls is realized by setting aside an initial section of the input buffer, which is then gradually filled up by the last n characters read. Here is an implementation of a class supporting series of `unget()` calls:

```

#include <unistd.h>
#include <streambuf>
#include <ios>
#include <algorithm>

class fdunget: public std::streambuf
{
public:
    fdunget (int fd, unsigned bufsz, unsigned unget)    // 9
    :
        fd(fd),
        reserved(unget)                                // 12
    {
        unsigned allocate =                                // 14
            bufsz > reserved ?
                bufsz
            :
                reserved + 1;

        buffer = new char [allocate];                    // 20

        base = buffer + reserved;                        // 22
        setg(base, base, base);                          // 23
    }
};

```

```

        bufsize = allocate - reserved;                // 25
    }

    ~fdunget()
    {
        delete buffer;
    }

    int
    underflow()
    {
        if (gptr() < egptr())
            return *gptr();

        unsigned consumed =                          // 39
            static_cast<unsigned>(gptr() - eback());

        unsigned move = std::min(consumed, reserved); // 41

        memcpy(base - move, egptr() - move, move);    // 43

        int nread = read(fd, base, bufsize);          // 45
        if (nread <= 0) // none read ->
            return EOF;

        setg(base - move, base, base + nread);        // 50

        return *gptr();
    }

private:
    int      fd;
    unsigned bufsize;
    unsigned reserved;
    char*    buffer;
    char*    base;
};

```

- At line 9 the constructor's definition starts. It expects as arguments a file descriptor, a buffer size and the number of characters which can be unget.
- At line 12 `unget` determines the size of a reserved area, which is the first reserved bytes of the used buffer.
- At line 14 we ensure that the number of bytes to allocate is at least one byte larger than reserved. So, a certain number of bytes will be read, and once reserved bytes have been read at least reserved bytes can be unget.
- At line 20 the buffer is allocated.
- At line 22 we set up the starting point for reading operations: it is called `base`, which is located reserved bytes into the buffer area. This is always the point where refills of the buffer start.
- At line 23 `streambuf`'s buffer pointers are set up. As no characters have been read as yet, all pointers are set to `base`. At the next read request, `underflow()` will now know that no characters are available, so `unget()` will fail immediately.

- At line 25 the size of the refill buffer is determined as the number of allocated bytes minus the size of the reserved area.
- At line 39 we are inside the member `underflow()`. Up to this point everything is standard, and now we determine how many characters were consumed. Remember that at line 23 we ascertained that the pointers indicated 0 bytes in the buffer, so `consumed` is either 0 or the actual number of bytes read into the buffer after a successful refill.
- At line 41 the number of bytes to move to the reserved area is computed. This number is at most `reserved`, but it is less if fewer characters were available. So, `move` is set to the minimum of the number of consumed characters and `reserved`.
- At line 43 the characters to move are moved from the end of the used section of the input buffer to the area immediately before `base`.
- Then, at line 45 the buffer is refilled. This is standard, but notice that reading starts from `base`, not from `buffer`.
- Finally, at line 50 `streambuf`'s read buffer pointers are set up. `Eback()` is set to move locations before `base`, thus defining the guaranteed unget-area, `gptr()` is set to `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 read character.

Here is an example of a program illustrating the use of the class `fdunget`. The program reads at most 10 characters from the standard input, stopping at EOF. A guaranteed unget-buffer of 2 characters is defined, in a buffer of 3 characters. Just before reading a character it is tried to unget at most 6 characters. This is of course never possible, but the program will neatly unget as many characters as possible, considering the actual number of characters read:

```
#include "fdunget.h"
#include <string>
#include <iostream>
#include <istream>

using namespace std;

int main(int argc)
{
    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
                    << "\tunget failed at attempt " << (ug + 1) << "\n"
                    << "\trereading: '";
```



```

        is.clear();
        while (ug--)
        {
            is.get(c);
            cout << c;

        }
        cout << "'\n";
        break;
    }
}

if (!is.get(c))
{
    cout << " reached\n";
    break;
}
cout << "Next character: " << c << endl;
}
}
/*
    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'
Next character: c
after reading 3 characters:
    unget failed at attempt 4
    rereading: 'abc'
Next character: d
after reading 4 characters:
    unget failed at attempt 4
    rereading: 'bcd'
Next character: e
after reading 5 characters:
    unget failed at attempt 4
    rereading: 'cde'
Next character:

after reading 6 characters:
    unget failed at attempt 4
    rereading: 'de'
,
    reached
*/

```

## 19.2 Using form() with ostream objects

In the ANSI/ISO standard does not include the previously available Gnu extension `form()` (and `scan()`) for stream objects. In this section the construction of a class `oformstream` is described, which is derived from `ostream` and does support `form()` and `vform()`. Analogously to `oformstream` a class `iscanstream` can be developed, supporting `scan()` and `vscan()`. The construction of this latter class is left as an exercise to the reader.

Here is the class interface and definition. It is annotated below:

```
#include <iostream>

class oformstream: public std::ostream
{
public:
    oformstream(std::ostream &ostr)
    :
        std::ostream(ostr.rdbuf())
    {}

    std::ostream &form(char const *fmt, ...);
};
```

- At line 6 the class is defined: it is derived from `ostream`, so it inherits `ostream`'s capabilities.
- At line 10 a simple constructor is defined. It expects the address of a `streambuf`, which could be obtained (as shown below) from another `ostream` object.
- At line 16 the member `form()` is defined. As is customary for this kind of function, it doesn't do the formatting itself, but delegates this to a 'v-function' companion, in this case `vform()`.
- At line 24 the function `vform()` is defined.
- At line 26 an initial estimate of the required buffer size for the formatting is defined. This initial estimate is set to a power of 2, so that eventually (line 41) it might 'enlarge' to 0.
- At line 28 formatting is attempted in iterations. At each iteration the formatting is attempted again, using a buffer that is twice as large.
- At line 30 an `auto_ptr` is used to manage the dynamically allocated buffer.
- At line 32 (33) the formatting is attempted. If it succeeds, `vsnprintf()` returns the number of bytes written to the buffer, not counting the final ASCII-Z character.
- At line 35, if the formatting succeeded, the contents of the buffer are written to the `ostream` base class, and the `oformstream` object is returned.
- Otherwise, at line 41, the size of the next buffer is determined. If this results in a zero-size, then an error message is written and the program terminates. More forgiving alternatives are left for the reader to develop.

A program using the class `oformstream` is given in the next example. It prints a well-known string and some marker text:

```
#include "oformstream.h"
```

```

#include <memory>

std::ostream &oformstream::form(char const *fmt, ...)
{
    va_list
        list;
    va_start(list, fmt);
    char
        c;
    unsigned
        n = vsnprintf(&c, 0, fmt, list);
    std::auto_ptr<char>
        buf(new char[n + 1]);

    vsprintf(buf.get(), fmt, list);

    return *this << buf.get();
}

using namespace std;

int main()
{
    oformstream
        of(cout);

    of << "Hello world\n";
    cout << "Ok, ";
    of << "That's all, folks\n";

    of.form("%s\n", "Hello world of C++") << endl;
};

```

## 19.3 Redirection revisited

Earlier, in section 5.8.3 it was noted that within a C++ program streams could be 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 realizing redirection at the level of the programming language itself.

It should be realized that this is fine within the context of the C++ program, but if that context is left, the redirection is terminated, as the operating system does not know about `streambuf` objects. This happens, e.g., when a program uses a `system()` call to start a subprogram. The following program uses C++ redirection to redirect the information inserted into `cout` to a file, and then calls

```
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 C++'s redirection would be recognized by the operating system. However, this is not the case, so `hello world` still appears at the program's standard output and not on the redirected file. A solution of this problem involves redirection at the operating system level, for which system calls like `dup()` and `dup2()` are available in some operating systems

(e.g., Unix and friends). Examples of the use of these system calls are given in section 19.4 in this chapter.

Here is the example of the *failing redirection* at the system level following C++ redirection using streambuf redirection:

```
#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace::std;

int main()
{
    ofstream
        of("outfile");

    cout.rdbuf(of.rdbuf());
    cout << "To the of stream" << endl;
    system("echo hello world");
    cout << "To the of stream" << endl;
}
/*
    Generated output: on the file 'outfile'
    To the of stream
    To the of stream
    On standard output:
    hello world
*/
```

## 19.4 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, but this requires the program to wait for the *child process* to terminate. The more general way to spawn subprocesses is to call `fork()`.

In this section we will see 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 will therefore focus on that operating system. However, other systems usually provide comparable facilities. The following discussion is based heavily on the notion of *design patterns*, as published by *Gamma et al.* (1995)

When `fork()` is called, the current program is duplicated in memory, thus creating a new process, and both processes continue their execution just below the `fork()` system call. The two processes may, however, inspect the return value of `fork()` which is different in the original process (called the *parent process*) and 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.

A basic Fork class should hide the bookkeeping details of a system call like `fork()` from its users. The Fork class developed below will do just that. The class itself only needs to take care of 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 is. However, a Fork object should be able to activate the child process.

In the child process created by the Fork class developed here, two subtasks will be performed: first, the necessary (standard) streams are redirected, second the actual child process is started.

Comparably, in the parent process two subtasks will also be performed: here `istream`s and an `ostream` object can be made available, followed by the activation of the actual parent process. However, the Fork process has no responsibilities here, but defers these tasks to separate objects. This *design pattern* is called the *Chain of Responsibility* pattern by Gamma *et al.* (1995). Using this design pattern the sender of a request is decoupled from its receiver.

Normally, the child process does not return: once it has completed its task it should terminate as a separate process. The parent (main) process usually waits for its child process to complete its task. The terminating child process informs its parent that it is about to terminate by sending a *signal* to the parent, which should be caught by the parent. If the child terminates and the parent process does not catch the signal generated by the child process then the child process remains visible as a so-called *zombi* process.

There is one situation where the *child* process continues to live, and the *parent* dies. In nature this happens all the time: parents tend to die before their children do. In our context this represents a *daemon* program, where the parent process dies and the child program continues in fact as a child of the basic `init` process. Again, when the child eventually dies a signal is sent to its 'step-parent' `init`. No *zombi* is created here, as `init` catches the termination signals of its (step-) children.

The construction of a daemon process is so simple, that no special Fork object is needed (see section 19.4.12 for an example).

### 19.4.1 The 'ChildProcess' abstract base class

The child process will be activated from a method offered by an abstract `ChildProcess` class, offering a protected pure virtual method `doProcess()` which must be implemented in a concrete implementation of the abstract class. Depending on the problem at hand, the child process may assume that it can read from its standard input, and that it can write to its standard output and/or standard error. Here is the definition of the abstract `ChildProcess` class:

```
#ifndef _CHILD_PROCESS_
#define _CHILD_PROCESS_

class ChildProcess
{
public:
    void process()
    {
        doProcess();
        exit(-1);           // doProcess itself should stop
    }

protected:
```

```

        virtual void doProcess() = 0;    // implemented in subclasses
    };

#endif

```

In this class, note that the method `process()` is the only *public* method that is offered. This method performs two tasks: it calls `doProcess()`, which is implemented in a derived class; and then, as a safeguard, it calls `exit(-1)`: this ensures that the child process is properly terminated if, for whatever reason, the `doProcess()` member itself fails to terminate the child process properly (see also section 19.4.12 for the reverse case where the child continues to live and the parent dies).

The design pattern in which a generic class defines a member (an action) to be implemented in concrete subclasses is called a *Command design pattern* by Gamma *et al.* (1995). The `process()` method defines a *skeleton algorithm*, deferring the implementation of the `doProcess()` step to a subclass. This design pattern is called the *Template Method* by Gamma *et al.*

### 19.4.2 The ‘ParentProcess’ abstract base class

The parent process acts comparably to the child process. The parent process will be activated from a method offered by an abstract `ParentProcess` class, offering a public pure virtual method `process()` which must be implemented in a concrete implementation of the abstract class. This `process()` method always is called with the following arguments:

- The `int pid`, being the *process ID* of the child process;
- A `std::ostream &out`, which may be used to write information to the child process (appearing at the child process’ standard input stream);
- A `std::istream &in`, which may be used to read information from the child process (e.g., written to the child process’ standard output stream);
- A `std::istream &inErr`, which may be used to read information from the child process (e.g., written to the child process’ standard error stream);

Here is the definition of the abstract class `ParentProcess`:

```

#ifndef _PARENT_PROCESS_
#define _PARENT_PROCESS_

#include <istream>

class ParentProcess
{
public:
    virtual void process          // pure virtual function
    (
        int pid,
        std::ostream &out,
        std::istream &in,
        std::istream &inerr
    ) = 0;
};

#endif

```

Again, this is an example of the *Command design pattern* as identified by Gamma *et al.* (1995).

### 19.4.3 The ‘Redirector’ abstract base class

Before the child process can receive information from the parent process and before the parent process can receive information from the child process, the proper communication channels must be constructed. The Fork class itself has no business here.

Redirection at the system level (and not at the C++ level, see section 19.3) is required to connect the child’s standard streams to the streams defined as parameters in the `ParentProcess::process()` member function. The abstract class `Redirector` defines the interface for this redirection. Depending on the requirements of the child process its standard input, standard output and/or standard error may have to be redirected. Here is the definition of the abstract class `Redirector`:

```
#ifndef _REDIRECTOR_H_
#define _REDIRECTOR_H_

#include <istream>

class Redirector
{
public:
    enum
    {
        R,
        W
    };

    enum bad_stream
    {};

    virtual ~Redirector()
    {}

    virtual void child() = 0;    // sets up redirections in the child
    virtual void parent() = 0;  // initializes in, out, err

    virtual std::istream &in()
    {
        throw bad_stream();
        return *reinterpret_cast<std::istream *>(0);
    }

    virtual std::istream &inErr()
    { return in(); }

    virtual std::ostream &out()
    {
        throw bad_stream();
        return *reinterpret_cast<std::ostream *>(0);
    }
};
```

```
#endif
```

This class offers the following methods:

- The pure virtual member `child()` defines the redirection of the necessary streams in the child process.
- The pure virtual member `parent()` defines the redirection of the necessary streams in the parent process.
- The virtual members `in()`, `inErr()` and `out()` throw by default `Redirector::bad_stream` exceptions. When a derived class actually defines a certain kind of redirection, it can override the default implementations to make the redirected streams available: `in()` should be made available to read the (standard) output of the child process, `inErr()` should be made available if the child's standard error must be read on a separate stream, `out()` should be made available to write information to the child's standard input.

The `Redirector` class represents yet another example of Gamma's (1995) *Command design pattern*.

#### 19.4.4 The 'Fork' class: implementation

Now that `Fork`'s support classes have been defined, let's have a look at the `Fork` class itself. The class defines but one member: `fork()`, performing the action `fork()` system call. An object of the class `Fork` can be *configured* by providing the appropriate `Redirector`, `ChildProcess` and `ParentProcess` objects. In the current definition the `Fork` constructor defines such parameters and the class defines corresponding reference variables, but an alternative approach where `fork()` itself defines these reference parameters could have been followed as well. It is up to the reader to experiment with this latter approach.

Here is the interface of the class `Fork`, together with the implementation of the `fork()` member function:

```
#ifndef _FORK_H_
#define _FORK_H_

class Redirector;
class ChildProcess;
class ParentProcess;

class Fork
{
public:
    Fork(Redirector &red, ChildProcess &cp, ParentProcess &pp)
    :
        redirector(red),
        child(cp),
        parent(pp)
    {}

    void fork() const;

protected:
    Redirector &redirector;
```



```

        ChildProcess &child;
        ParentProcess &parent;
};

#endif

#include "fork.i"

void Fork::fork() const
{
    int pid = ::fork();
    if (pid == 0)           // child
    {
        redirector.child();
        child.process();
    }

    redirector.parent();
    parent.process(pid,
                    redirector.out(),
                    redirector.in(),
                    redirector.inErr());
}

```

Looking at the implementation of the `fork()` member, note that its `child-` and `parent-` sections are implemented rather symmetrically: first the `redirector` is called to set up the appropriate redirection, then the relevant (`child` or `parent`) object is asked to do its thing.

Again we meet a *design pattern* as identified by Gamma *et al.* (1995), this time the *Chain of Responsibility* pattern (see also section 19.4).

### 19.4.5 Support classes

Now that the groundwork has been completed, we turn our attention to the construction of concrete examples. In order to use the `Fork` class, the abstract base classes `Redirector`, `ChildProcess` and `ParentProcess` must be implemented in concrete subclasses. The following classes will be developed:

- The class `IORedirector` implements the redirection of the standard input stream, standard output stream and standard error stream in a `Fork` object. Rather than redirecting the standard error and standard output to different streams, they will be merged to one stream, which will be read by the parent via the `redirector.in()` istream.
- The class `ChildPlain` will allow us to activate a simple child process, not expecting any arguments, producing output at its standard output and/or standard error stream and maybe also expecting input at its standard input stream.
- The class `ParentCmd` is used to communicate with a child process by sending commands to the child process (arriving at the child's standard input stream) and reading the child's standard output or error in return.
- The class `ParentSlurp` is used just to read the child's standard output or error.

Redirection at the system level is accomplished using the `dup2()` system call, operating on *file descriptors*. One of these file descriptors is the end of a *pipe*, as created by the `pipe()` system call. A pipe is a one-way channel through which information may flow. Typically one process writes information to the pipe, another process reads information from the pipe.

The `pipe()` system call will fill an `int` array having two elements with two file descriptors: element 0 represents the end of the pipe from which information can be *read*, element 1 represents the end to which information can be written.

The `Redirector` subclasses must set up the redirection, and this is often a rather standard initialization and redirection of the file descriptors created by a `pipe()` system call. Because this is often standard, it is useful to have a set of objects available doing just that. So, as an extra support class to assist the implementation of `Redirector` subclasses the `Pipe` set of classes were developed too.

The abovementioned classes are now constructed. First the `Pipe` set of classes are developed, followed by the construction of the `IORedirector`, the `ChildPlain`, the `ParentCmd` and the `ParentSlurp` classes. Eventually, these classes are used in some examples.

Although the story seems like a long one, it is hoped that the reader will be able to appreciate the beauty of the design patterns underlying the construction of the `Fork`- and its support-classes as discussed here. As we will see, once the design is clear, the implementations are surprisingly short.

### 19.4.6 The ‘Pipe’ classes

Three `Pipe` support classes were constructed: `Pipe` is the base class, defining and initializing the file descriptor array. Its constructor is used only by its subclasses, so it was given the `protected` access modifier. Here is the definition of the class `Pipe`:

```
#ifndef _PIPE_H_
#define _PIPE_H_

#include <unistd.h>

class Pipe
{
public:
    enum bad_pipe {};

    int getReadFd()
    {
        return fd[R];
    }
    int getWriteFd()
    {
        return fd[W];
    }
protected:
    enum { R, W };

    Pipe()
    {
        if (pipe(fd))
            throw bad_pipe();
    }
};
```

```

        int fd[2];
};

#endif

```

The IPipe class defines a pipe which can be used by a parent process to read from an ifdnstreambuf object (introduced in section 19.1.2). The redirection of a (standard) file descriptor to the file descriptor corresponding to the writing end of the pipe is implemented in its child() members. Note that the unused ends of the pipes (the reading end in the child, the writing end in the parent) are closed: open descriptors will prevent the corresponding pipe to close if only one of the processes using the pipe is done with it. Also, once the redirection has been performed (to save space the successful completion of dup2() has not been checked: it should be checked in production code), the file descriptor associated with the pipe can be closed as well, as the alternate file descriptor will now be used instead. Also note the use of the open() member function of the ifdnstreambuf object. Here is the definition of the IPipe class:

```

#ifndef _IPIPE_H_
#define _IPIPE_H_

#include "pipe.h"
#include "../examples/ifdnbuf.h"

class IPipe: public Pipe           // parent reads
{
public:
    void child(int std_fd)
    {
        close(fd[R]);
        dup2(fd[W], std_fd);
        close(fd[W]);
    }

    void child(int std_fd1, int std_fd2)
    {
        close(fd[R]);
        dup2(fd[W], std_fd1);
        dup2(fd[W], std_fd2);
        close(fd[W]);
    }

    void parent(ifdnstreambuf &fdbuf, unsigned bufsize = 500)
    {
        close(fd[W]);
        fdbuf.open(fd[R], bufsize);
    }
};

#endif

```

The OPipe class defines a pipe which can be used by a parent process to write to an ofdnstreambuf object (introduced in section 19.1.1). This information will reach the reading end of the pipe in the child process. The implementation of IPipe closely mimics the implementation of the class IPipe. Since there is only one standard input stream, the child() member needs no parameter, but can immediately redirect STDIN\_FILENO. Here is the definition of the class OPipe:

```

#ifndef _OPIPE_H_
#define _OPIPE_H_

#include "pipe.h"
#include "../examples/fdout.h"

class OPipe: public Pipe           // parent writes
{
    public:
        void child()
        {
            close(fd[W]);
            dup2(fd[R], STDIN_FILENO);
            close(fd[R]);
        }

        void parent(ofdnstreambuf &fdbuf, unsigned bufsize = 500)
        {
            close(fd[R]);
            fdbuf.open(fd[W], bufsize);
        }
};

#endif

```

### 19.4.7 The ‘IORedirector’

The class `IORedirector`, derived from the `Redirector` abstract base class, redirects information written in a parent process to the writing end of a pipe, accessed through its `out()` ostream to the standard input stream of a child process that uses the reading end of the pipe. Furthermore, any information that's written in the child process to either the standard output stream or the standard error stream may be read in the parent process from the reading end of a pipe, which is accessed through the istream `in()` reference. First the class definition is shown, then it will be annotated. Here is the class definition:

```

#ifndef _IOREDIRECTOR_H_
#define _IOREDIRECTOR_H_

#include "../abc/redirector.h"
#include "../pipe/ipipe.h"
#include "../pipe/opipe.h"

class IORedirector: public Redirector    // defines in() out()
{
    public:
        IORedirector()                  // 11
        :
            ins(&ifdb),
            outs(&ofdb)
        {}

        std::istream &in()              // 17
        {

```

```

        return ins;
    }

    std::ostream &out()          // 22
    {
        return outs;
    }

    void child()                 // 27
    {
        ip.child(STDOUT_FILENO, STDERR_FILENO);
        op.child();
    }

    void parent()                // 33
    {
        ip.parent(ifdb);
        op.parent(ofdb);
    }

private:
    ifdnstreambuf ifdb;          // 40
    std::istream ins;
    IPipe ip;

    ofdnstreambuf ofdb;
    std::ostream outs;
    OPipe op;                   // 47
};

#endif

```

- Looking at the private data section, we see (lines 40 to 47) a `ifdnstreambuf` (see section 19.1.2), an `istream` and an `IPipe` input-pipe (see section 19.4.6) which is used for reading information from the child process. A comparable set of objects is defined for writing information to the child process.
- The constructor (line 11) merely associates the `istream` and `ostream` objects with their respective buffers, constructed by their default constructors.
- The `in()` and `out()` members (lines 17 and 22) override the exception throwing default `Redirector` implementations.
- Realize that in the child process the parent's input pipe becomes a writing end, and vice versa. So, the `IPipe ip` in the child process redirects the standard output and standard error streams to that pipe. The input and output pipes are initialized in the `child()` member defined in line 27.
- In the parent process, the pipes must be associated with the `streambuf` objects. This is realized by the `parent()` member, defined in line 33.

Note that `IORedirector` can be used in many situations. If a process only writes information, the reading facilities are simply not used. If a process only reads information, the writing facilities are not used. The class does not distinguish information written by the child process to its standard output stream from information written to its standard error stream. If these streams

must be distinguished, another subclass of `Redirector` must be constructed (probably requiring the use of the `select()` system call in the parent). This situation is left as an exercise to the reader.

### 19.4.8 The 'ChildPlain' class

The class `ChildPlain`, derived from the `ChildProcess` abstract base class, only has to define our child process. The current implementation (see below) defines a public constructor, expecting the full path of the program to execute as child process. In the current context, it is assumed that the program to be executed needs no arguments. So, the program could be `/bin/sh` or `/bin/ls`.

The member `doProcess()` is never seen outside of the `ChildProcess` class itself, so it needs no public access rights. Here, it is defined as a private member.

At the very minimum, only `doProcess()` could have been defined, using a fixed call of the process. For illustration purposes `ChildPlain` was constructed as a slightly more flexible class. Here is the class definition itself:

```
#ifndef _CHILD_PLAIN_
#define _CHILD_PLAIN_

#include "../abc/childprocess.h"

#include <unistd.h>

class ChildPlain: public ChildProcess
{
public:
    ChildPlain(char const *cmd)
    :
        cmd(cmd)
    {}
private:
    void doProcess()
    {
        execl(cmd, cmd, 0);
    }
    char const *cmd;
};

#endif
```

### 19.4.9 The 'ParentCmd' class

The class `ParentCmd`, derived from the `ParentProcess` abstract base class, only needs to implement the `process()` member, defining the parent's side of the communication with the child process. `ParentCmd` was constructed with the child process executing `/bin/sh` in mind. The `process()` member implements the following simple protocol:

- Lines are read from `cin`, reading stops at an empty line.

- Each line is inserted into the ostream object referenced by the out parameter. An echo EOC (End Of Command) is appended, thus defining a very simple protocol.
- All output is read from the istream object referenced by the in parameter. Once EOC is sensed, reading stops.
- Once no more lines are read from cin, the child process may terminate. This is realized by sending a SIGTERM signal to the child process, followed (to be sure the process terminates) by the sending of a SIGKILL signal.
- Eventually, the parent process prevents the occurrence of a zombi process by waiting for the child process to terminate using the waitpid() system call.

Here is the definition of the ParentCmd class.

```
#ifndef _PARENT_CMD_
#define _PARENT_CMD_

#include "../abc/parentprocess.h"

#include <iostream>
#include <string>
#include <sys/wait.h>
#include <signal.h>

using namespace std;

class ParentCmd: public ParentProcess
{
public:
    void process
    (
        int pid,
        std::ostream &out,
        std::istream &in,
        std::istream &inerr    // not used
    )
    {
        string
            s;

        while (true)
        {
            cerr << "? ";
            if (!getline(cin, s))
                break;

            cout << "Cmd: " << s << endl;
            out << s << endl << "echo EOC" << endl;

            while (getline(in, s))
            {
                cout << "Read: " << s << endl;
                if (s == "EOC")
                    break;
            }
        }
    }
};
```

```

        }
    }
    kill(pid, SIGTERM);
    kill(pid, SIGKILL);

    waitpid(pid, 0, 0);
}

};

#endif

```

### 19.4.10 The ‘ParentSlurp’ class

The class `ParentSlurp`, derived from the `ParentProcess` abstract base class, only needs to implement the `process()` member, defining the parent's side of the communication with the child process. `ParentSlurp` was constructed with the child process executing a program merely producing output (like `/bin/sh`) in mind. The `process()` member reads all information appearing at its `istream` `&in` parameter (note the use of the C++ redirection), and then waits for the child process to complete.

Here is the definition of the `ParentSlurp` class:

```

#ifndef _PARENT_SLURP_
#define _PARENT_SLURP_

#include "../abc/parentprocess.h"

#include <iostream>
#include <string>
#include <sys/wait.h>
#include <signal.h>

using namespace std;

class ParentSlurp: public ParentProcess
{
public:
    void process
    (
        int pid,
        std::ostream &out,          // not used
        std::istream &in,
        std::istream &inerr         // not used
    )
    {
        std::cout << in.rdbuf();
        waitpid(pid, 0, 0);
    }
};

#endif

```



### 19.4.11 The 'Fork' class: example of use

Now that concrete classes have been derived from the abstract base classes, it's time to construct some examples using the Fork class. Two examples are presented here. Both examples use the IORedirector object (introduced in section 19.4.7).

In the first example, /bin/ls is called as child process, and a ParentSlurp object is used to consume the generated output:

```
#include "fork/fork.h"
#include "ioredirector/ioredirector.h"
#include "childplain/childplain.h"
#include "parentslurp/parentslurp.h"

int main()
{
    IORedirector    io;
    ChildPlain      binls("/bin/ls");
    ParentSlurp     slurp;

    Fork(io, binls, slurp).fork();
}
/*
    Generated output (partially shown):
abc
binls
binsh
...
parentcmd
parentslurp
pipe
*/
```

Note the combined construction of the Fork object as a constant object, and the way fork() is called with this constant object. Alternatively, the construction of the Fork object and the activation of the fork() member could have been separated:

```
Fork    f(io, binsh, cmd);
f.fork();
```

In the second example, /bin/sh is called to execute several commands, and a ParentCmd object is used to interface between the user of the program and the /bin/sh program. Here is the program and an example of a resulting conversation:

```
#include "fork/fork.h"
#include "ioredirector/ioredirector.h"
#include "childplain/childplain.h"
#include "parentcmd/parentcmd.h"

int main()
{
    IORedirector    io;
    ChildPlain      binsh("/bin/sh");
```

```

        ParentCmd      cmd;

        Fork(io, binsh, cmd).fork();
    }
    /*
        Example of conversation:
    ? cd tmp
    Cmd: cd tmp
    Read: /bin/sh: cd: tmp: No such file or directory
    Read: EOC
    ? mkdir tmp
    Cmd: mkdir tmp
    Read: EOC
    ? cd tmp
    Cmd: cd tmp
    Read: EOC
    ? cd ../
    Cmd: cd ../
    Read: EOC
    ? rmdir tmp
    Cmd: rmdir tmp
    Read: EOC
    ? cd tmp
    Cmd: cd tmp
    Read: /bin/sh: cd: tmp: No such file or directory
    Read: EOC
    */

```

### 19.4.12 The ‘Daemon’ program

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 following 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 uses none of the features currently offered by the `Fork` class.

Here is an example illustrating the construction of a daemon program:

```

#include <iostream>
#include <unistd.h>

int main()
{
    int pid = fork();
    if (pid != 0)
        return pid < 0;    // 1: failing fork, 0: daemon started

    // child process: the daemon

```

```

        sleep(3);                // wait 3 seconds
        std::cout << "Hello from the child process\n";
    }
    /*
    Generated output:
    The next command prompt, then after 3 seconds:
    Hello from the child process
    */

```

## 19.5 Function objects performing bitwise operations

In section 17.1 several types of predefined function objects were introduced. Predefined function objects exist performing arithmetic operations, relational operations, and logical operations exist, corresponding to a multitude of binary- and unary operator unary operators.

For whatever reason, some operators are missing: there are 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 template class implementing a function object calling the bitwise and (`operator&()`), and a template class implementing a function object calling the unary not (`operator~()`). Other missing operators can be constructed similarly. This is, however, left as an exercise to the reader.

Here is the implementation of a function object calling the bitwise `operator&()`:

```

#include <functional>

template <typename _Tp>
struct bit_and : public binary_function<_Tp,_Tp,_Tp>
{
    _Tp operator()(const _Tp& __x, const _Tp& __y) const
    {
        return __x & __y;
    }
};

```

Here is the implementation of a function object calling `operator~()`:

```

#include <functional>

template <typename _Tp>
struct bit_not : public unary_function<_Tp,_Tp>
{
    _Tp operator()(const _Tp& __x) const
    {
        return ~__x;
    }
};

```

These and other missing predefined function objects are also implemented in the file `bitfunctional`, which is found in one of the `cplusplus` zip-archives.

An example using `bit_and()` to remove all odd numbers from a vector of `int` values, using the `remove_if()` generic algorithm is:

```

#include <iostream>
#include <algorithm>
#include <vector>
#include "bitand.h"

int main()
{
    vector<int>
        vi;

    for (int idx = 0; idx < 10; ++idx)
        vi.push_back(idx);

    copy
    (
        vi.begin(),
        remove_if(vi.begin(), vi.end(), bind2nd(bit_and<int>(), 1)),
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;
}
/*
    Generated output:
    0 2 4 6 8
*/

```

## 19.6 Implementing a reverse\_iterator

In this section we will extend the class `Vector`, introduced in section 18.3 to contain a `RandomAccessIterator reverse_iterator`.

First, we note that applying the ordinary iterators of the `Vector` class (made available by the `begin()` and `end()` member functions) to the generic algorithms is without any problem, as the pointers that are returned by these members are accepted by the generic algorithms:

```

#include <iostream>
#include "vector.h"

int main()
{
    Vector<int>
        vi;

    for (int idx = 0; idx < 10; ++idx)
        vi.push_back(idx);

    random_shuffle(vi.begin(), vi.end());

    copy(vi.begin(), vi.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}

```

```

/*
    generated output, e.g.,
    1 7 9 3 5 6 0 4 8 2
*/

```

However, the use of a reverse iterator is not without problems: the reverse iterator shows rather complex behavior, in that it visits the elements in its range backwards, resulting in a reversal of the meanings of the operators that are used to step forward and step backward in a range. If a reverse iterator `rit` points to element 11 in a range, then it will point to element 10, rather than 12 after `++rit`. Because of this complex behavior, the reverse iterator will be implemented as a class `reverse_iterator`, which can do the required bookkeeping for us.

Furthermore, in the current context `reverse_iterator` is a reverse iterator *only* for the elements of the class `Vector`. So, a *nested class* seems to be appropriate here. Since `Vector` has already been defined, we will decide to leave it untouched. Instead of modifying `Vector`, a template class `Vector2` is derived of it, and `Vector2` will be given its nested class `reverse_iterator`. This latter class can be derived from `random_access_iterator` to realize a `RandomAccessIterator` for the class `Vector2`:

```

template <typename Type>
class Vector2: public Vector<Type>
{
    public:
        class reverse_iterator:
            public random_access_iterator<Type, ptrdiff_t>
        {
            friend class Vector2<Type>;    // see the text, below
            ...
        };
        ...
};

```

The `ptrdiff_t` type that is mentioned here is ordinarily defined as an `int` (e.g., with the Linux operating system it is defined as a `_kernel_ptrdiff_t` in `types.h`, which is defined as an `int` in `posix_types.h`).

The `reverse_iterator` class is now defined as a nested template class, derived from the template class `random_access_iterator`. Its surrounding class is `Vector2`, which itself is a template class that was derived from the template class `Vector`.

Within `reverse_iterator` we store two `Type` pointers, called `begin` and `end`, which will be used to define the reversed range `[end - 1, begin - 1)`. In the private section of `reverse_iterator` we declare:

```

Type
    *begin,
    *end;

```

`begin` and `end` receive their initial values from the `begin()` and `end()` members of the `Vector` object. However, the class `reverse_iterator` is closely tied to `Vector2`: only `Vector2` objects should be allowed to construct their own reverse iterators. This policy is enforced by making `Vector2<Type>` a *bound friend* of `reverse_iterator`, and by declaring its constructor private. The constructor expects the two `Type` pointers, defining the range `[begin(), end())`. Internally `begin() - 1` will be used as *endpoint* of the reversed range, while `end() - 1` will be used as the beginning of the

reversed range. The constructor, defined in the private section of the `reverse_iterator` class, is therefore:

```
reverse_iterator(Type *begin, Type *end)
:
    begin(begin),
    end(end)
{}
```

In the implementation of `reverse_iterator`, `end` will gradually walk towards `begin`, which will keep its value. The *dereferencing operator* will not return `*begin`, but rather `end[-1]`: the element just before the element to which `end` points. All this is allowed, as `Vector::begin()` and `Vecotr::end()` return random access iterators themselves, for which pointer arithmetic operations are defined.

Since the `reverse_iterator` class declares `Vector2<Type>` as a friend, the `Vector2` class can implement two members, `rbegin()` and `rend()` returning `reverse_iterator` objects:

```
reverse_iterator rbegin()
{
    return reverse_iterator(begin(), end());
}
reverse_iterator rend()
{
    return reverse_iterator(begin(), begin());
}
```

Apart from these two members, the `Vector2` class could define two constructors, analogous to the constructors in the `Vector` class. E.g.,

```
Vector2()
:
    Vector<Type>()
{}
```

In order to allow programs to use these reverse iterators, a public copy constructor and overloaded assignment operator is needed in the class `reverse_iterator`. Since the `Type` pointers in `reverse_iterator` are merely used for pointing to existing information, a destructor is not required. Of course, this implies that the usefulness of a `reverse_iterator` terminates if the object from which it was obtained goes out of scope. It is the responsibility of the programmer to make sure this doesn't cause disaster. In the public section of `reverse_iterator` we add the following members (implementations given as inline code):

```
reverse_iterator(reverse_iterator const &other)
:
    begin(other.begin),
    end(other.end)
{

reverse_iterator &operator=(reverse_iterator const &other)
{
    begin(other.begin);
    end(other.end);
}
```

The implementation shown below is fairly Spartan: illegal pointer values are not detected. As the generic algorithms do not require these checks we've left them out. Of course, one could modify the implementation and incorporate checks at several locations. It's up to the reader.

Random access iterators require us to implement a series of member functions. Here they are, as they should appear in the public section of the `reverse_iterator` nested class. Most implementations will be self-explanatory, and consist of single lines of code.

```
Type &operator*() const
{
    return end[-1];          // reference to the previous element
}

bool operator==(reverse_iterator const &other) const
{
    return other.end == end;  // comparing two iterators for equality
}

reverse_iterator &operator++()
{
    --end;                   // increment is: to the previous element
    return *this;
}

reverse_iterator &operator--();
{
    ++end;                   // decrement is: to the next element
    return *this;
}

// remaining members: see the text
bool operator<(reverse_iterator const &other) const;

reverse_iterator operator++(int);
reverse_iterator operator--(int);

int operator-(reverse_iterator const &other) const;
reverse_iterator operator+(int step) const;
reverse_iterator operator-(int step) const;
```

The following member deserve special attention:

- `bool operator<(reverse_iterator const &other) const`

This function compares two iterators. The `operator<()` must be interpreted as 'the lvalue iterator points to an element earlier in the range than the rvalue iterator'. Since we're talking *reversed iterators* here, this means that the lvalue operator points to a *later* element than the rvalue operator in the normal iterator range. So, the overloaded operator should return true if `this->end > other.end`, implying that the current iterator points to an element that is located earlier in the range than the other iterator. So, we implement:

```
    return end > other.end;
```

- `reverse_iterator operator++(int)`

The postfix increment operator must return a `reverse_iterator` pointing to the current element, incrementing the pointer. Again, this implies that `end` is *decremented*. Since we have a constructor expecting two (normal) iterators,, the implementation again is a one-liner:

```
return reverse_iterator(begin, end--);
```

Using the postfix decrement with `end`, a `reverse_iterator` object is returned representing the iterator's position at the time the postfix increment operator was called, incrementing (so, decrementing `end`) the reversed iterator as a side effect.

- `reverse_iterator operator++(int)`

Implemented analogously:

```
return reverse_iterator(begin, end++);
```

- `int operator-(reverse_iterator const &other) const`

This is the pointer arithmetic operator: it returns the difference between two pointers, representing the number of steps to make to reach the lvalue operand starting from the rvalue operand. Normally we would implement this as `end - other.end`. However, since reversed iterators *reduce* their values to reach elements *later* in the range, the algebraic sign must be negated with reversed iterators: `-(end - other.end)`, or, using simple algebraic manipulations:

```
return other.end - end;
```

- `reverse_iterator operator-(int step) const`

This operator allows us to use random steps backward using a random access iterator. With a reversed iterator this means that we must increase the value of the iterator. So we return the following `reverse_iterator` object: `return reverse_iterator(begin, end + step);`

- `reverse_iterator operator+(int step) const`

This operator does the opposite. So we implement it accordingly: `return operator+(-step);`

For illustration purposes we show how to implement some of these member functions outside of the template class interface: these implementations could be included in the header file defining the `Vector2` interface, if the corresponding member functions are only declared:

```
template <typename Type>                                // this is a template function
                                                         // the class in which the
Vector2<Type>::reverse_iterator<Type>:: // constructor is defined
reverse_iterator(Type *begin, Type *end)// the constructor itself
:
    begin(begin),
    end(end)
{}

template <typename Type>                                // this is a template function
Vector2<Type>::reverse_iterator                        // this is its return type
                                                         // the class in which the function
Vector2<Type>::reverse_iterator<Type>:: // is defined
operator+(int step) const                             // this is the function itself
{
```



```

        return reverse_iterator(begin, end - step);
    }

```

Finally, an example of a program using the `reversed_iterator`:

```

#include <iostream>
#include "vector3.h"

int main()
{
    Vector2<int>
        vi;

    for (int idx = 0; idx < 10; ++idx)
        vi.push_back(idx);

    copy(vi.begin(), vi.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    random_shuffle(vi.rbegin(), vi.rend());
    copy(vi.begin(), vi.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    sort(vi.rbegin(), vi.rend());
    copy(vi.begin(), vi.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    reverse(vi.rbegin(), vi.rend());
    copy(vi.begin(), vi.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    copy(vi.rbegin(), vi.rend(), ostream_iterator<int>(cout, " "));
    cout << endl;
}
/*
    Generated output:
0 1 2 3 4 5 6 7 8 9
7 1 5 9 3 4 6 0 2 8
9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
*/

```

## 19.7 Using Bison and Flex

The example discussed in this section digs into the peculiarities of using a parser- and scanner generator with C++. Once the input for a program exceeds a certain level of complexity, it's advantageous to use a scanner- and parser-generator for creating the code which does the actual input recognition. The current example assumes 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'.

However, scanner- and parser generators are also (and maybe even more commonly, nowadays) available as free software. Both `bison` and `flex` can be obtained from `ftp://prep.ai.mit.edu/pub/gnu`. `Flex` will create a C++ class when called as `flex++`, or when the `-+` flag is used. With `bison` the situation is a bit more complex. Scattered over the Internet several `bison++` archives can be found (e.g., in `ftp://ftp.rug.nl/contrib/frank/software/linux/bison/` or `rzbsdi01.uni-trier.de`). The information in these archives usually dates back to 1993, irrespective of the version number mentioned with the archive itself.

Using `flex++` and `bison++` a class-based scanner and parser can be generated. The advantage of this approach is that the interface to the scanner and the parser tends to become a bit cleaner than without the class interface.

Below two examples are given. In the first example only a lexical scanner is used to monitor the production of a file from several parts. This example focuses on the lexical scanner, and on switching files while churning through the information. The second example uses both a scanner and a parser to transform standard arithmetic expressions to their postfix notation, commonly encountered in code generated by compilers and in HP-calculators. The second example focuses on the parser.

### 19.7.1 Using Flex++ to create a scanner

In this example a lexical scanner is used to monitor the production of a file from several subfiles. This example focuses on the lexical scanner, and on switching files while churning through the information. The setup is as follows: The input-language knows of an `#include` statement, which is followed by a string indicating the file which should be included at the location of the `#include`.

In order to avoid complexities that have 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 should terminate using a proper 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 uses a maximum nesting depth. Once the maximum is exceeded, the program terminates with an appropriate error message. In that case, the filename stack indicating where which file was included should be printed.

One small extra feature is that comment-lines should be recognized: `include` directives in comment-lines should be ignored, comment being the standard C++ comment-types.

The program is created in the following steps:

- First, the file `lexer` is constructed, containing the specifications of the input-language.
- From the specifications in `lexer` the requirements for the class `Scanner` evolve. The `Scanner` class is a wrapper around the class `yyFlexLexer` generated by `flex++`. The requirements results in the specification of the interface for the class `Scanner`.
- Next, the `main()` function is constructed. A `Startup` object is created to inspect the command-line arguments. If successful, the scanner's member `yylex()` is called to construct the output file.
- Now that the global setup of the program has been specified, the member functions of the different classes are constructed.
- Finally, the program is compiled and linked.

## The flex++ specification file

The organization of the lexical scanner specification file is similar to the one used with `flex`. However, `flex++` creates a class (`yyFlexLexer`) from which the class `Scanner` will be derived.

The code associated with the regular expression rules will be located inside the class `yyFlexLexer`. However, it would be handy to access the member functions of the derived class within that code. Fortunately, inheritance helps us to realize this. In the specification of the class `yyFlexLexer()`, we notice that the function `yylex()` is a *virtual* function. In the `FlexLexer.h` header file we see `virtual int yylex()`:

```
class yyFlexLexer: public FlexLexer
{
    public:
        yyFlexLexer( istream* arg_yyin = 0, ostream* arg_yyout = 0 );

        virtual ~yyFlexLexer();

        void yy_switch_to_buffer( struct yy_buffer_state* new_buffer );
        struct yy_buffer_state* yy_create_buffer( istream* s, int size );
        void yy_delete_buffer( struct yy_buffer_state* b );
        void yyrestart( istream* s );

        virtual int yylex();
        virtual void switch_streams( istream* new_in, ostream* new_out );
};
```

Consequently, if `yylex()` is defined in a derived class, then this function of the derived class will be called from a base class (i.e., `yyFlexLexer`) pointer. Since the `yylex()` function of the derived class is called, that function will have access to the members of its class, and to the public and protected members of its base class.

The context in which the generated scanner is placed is (by default) the function `yyFlexLexer::yylex()`. However, this context can be changed by defining the `YY_DECL`-macro. This macro, if defined, determines the context in which the generated scanner will be placed. So, in order to make the generated scanner part of the *derived class* function `yylex()`, three things must be done:

- The macro `YY_DECL` must be defined in the lexer specification file. It must define the derived class function `yylex()` as the scanner function. For example:

```
#define YY_DECL int Scanner::yylex()
```

- The function `yylex()` must be declared in the class definition of the derived class.
- As the function `yyFlexLexer::yylex()` is a *virtual* function, it must still be defined. It is not called, though, so its definition may be a simple

```
int yyFlexLexer::yylex()
{
    return 0;
}
```

The definition of the `YY_DECL` macro and the `yyFlexLexer::yylex()` function can conveniently be placed in the lexer specification file, as shown below.

Looking at the regular expressions themselves, notice that we'll need rules for the recognition of the comment, for the `#include` directive, and for the remaining characters. This is all fairly standard practice. When an `#include` directive is detected, the directive is parsed by the scanner. This is common practice: preprocessor directives are normally 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 `Scanner::Error` value (`invalidInclude`) if this fails. If the filename could be extracted, it is stored in `nextSource`, and on completion of the reading of the `#include` directive `switchSource()` is called to perform the switch to another file.

When the end of the file (EOF) is detected, the derived class' member function `popSource()` is called, which will pop the previously pushed file, returning `true`. Once the file-stack is empty, the function will return `false`, resulting in the call of `yyterminate()`, which will terminate the scanner.

The lexical scanner specification file has three sections: a C++ *preamble*, containing code which can be used in the code defining the actions to be performed once a regular expression is matched, a Flex++ *symbol area*, which is used for the definition of symbols, like a mini scanner, or *options*, like `%option yylineno` when the lexical scanner should keep track of the line numbers of the files it is scanning and, finally a *rules section*, in which the regular expressions and their actions are given. In the current example, the main task of the lexer is to copy information from the `istream *yyin` to the `ostream *yyout`, for which the predefined macro `ECHO` can be used. The complete and annotated lexical scanner specification file to be used with flex++ is:

```
%{
/* -----
                                C++ -preamble.
    Include header files, other than those generated by flex++ and bison++.
    E.g., include the interface to the class derived from yyFlexLexer
    -----*/

                                // the yylex() function that's actually
                                // used
#define YY_DECL int Scanner::yylex()

#include "scanner.H"           // The interface of the derived class

int yyFlexLexer::yylex()      // not called: overruled by
{                             // Scanner::yylex()
    return 0;
}

%}

/* -----
                                Flex++ symbol area
                                ~~~~~
    The symbols mentioned here are used for defining e.g., a mini scanner
    ----- */
%x comment
%x include
%option yylineno

eolnComment    "//".*
anyChar        .|\n

/* -----
```

#### Flex rules area:

```
-----
Regular expressions below here define what the lexer will recognize.
----- */

%%

/*
   The comment-rules: comment lines are ignored.
*/
{eolnComment}
"/*"          BEGIN comment;
<comment>{anyChar}
<comment>"*/"  BEGIN INITIAL;

/*
   File switching: #include <filepath>
*/
#include[ \t]+"<"  BEGIN include;
<include>[^ \t>]+  nextSource = yytext;
<include>">"[ \t]*\n {
    BEGIN INITIAL;
    performSwitch();
}
<include>{anyChar} throw invalidInclude;

/*
   The default rules: eating all the rest, echoing it to output
*/
{anyChar}      ECHO;

/*
   The <<EOF>>)rule: pop a pushed file, or terminate the lexer
*/
<<EOF>>        {
                if (!popSource())
                    yyterminate();
                }

```

Since the derived class is able to access the information stored within the lexical scanner itself (it can even access the information *directly*, since the data members of `yyFlexLexer` are protected, and thus accessible to derived classes), very much processing can be done by the derived class' member functions. This results in a very clean setup of the lexer specification file, which requires hardly any code in the *preamble*.

#### The derived class: Scanner

The class `Scanner` is derived from the class `yyFlexLexer`, generated by `flex++`. The derived class has access to the data controlled by the lexical scanner. In particular, the derived class has access to the following data members:

- `char *yytext`: contains the text matched by a regular expression (accessible outside of the scanner from the `YYText()` member);

- `int yyleng`: the length of the text in `yytext` (accessible outside of the scanner from the `YYLeng()` member);
- `int yylineno`: the current line number (only if %option `yylineno` was specified in the lexer specification file, accessible outside of the scanner from the `lineno()` member);

Other members are available as well, but we feel they are less often used. Details can be found in the file `FlexLexer.h`, which is part of the `flex` distribution.

The class `Scanner` performs two tasks: It pushes file information about the current file to a file stack, and pops the information pushed last once EOF is detected in a file.

Several member functions are needed to accomplish these tasks. As they are auxiliary to the scanner, they are private members. In practice, these private members are developed once the need for them arises. In the following interface of the `Scanner` class the final header file is given. Note that, apart from the private member functions, several private data members are used as well. These members are initialized by the `Scanner()` constructor and are used in the private member functions. They are discussed below, in the context of the member functions that are using them. Here is the `scanner.h` header file:

```
#include <FlexLexer.h> // provides yyFlexLexer interface
#include <fstream>
#include <string>
#include <stack>
#include <vector>

using namespace std;    // FBB: needed for g++ >= 3.0

class Scanner: public yyFlexLexer
{
public:
    enum Error
    {
        invalidInclude,
        circularInclusion,
        nestingTooDeep,
        cantRead,
    };

    Scanner(istream *yyin, string const &initialName);
    string const &lastFile()
    {
        return fileName.back();
    }
    void stackTrace(); // dumps filename stack to cerr
    int yylex();       // overruling yyFlexLexer's yylex()
private:
    Scanner(Scanner const &other);    // no Scanner copy-initialization
    Scanner &operator=(Scanner const &other); // no assignment either

    bool popSource(); // true: source popped
    void performSwitch();
    void throwOnCircularInclusion();

    stack<yy_buffer_state *>
```

```

        state;
vector<string>
    fileName;
string
    nextSource;

static unsigned const
    sizeof_buffer = 16384,
    maxDepth = 10;
};

```

If the scanner can extract a filename from an `#include` directive, a switch to another file is performed by `performSwitch()`. If the filename could not be extracted, the scanner throws an `invalidInclude` exception value. The `performSwitch()` member and the matching function `popSource()` handle the file switch. In particular, note that `yylineno` is not updated when a file switch is performed. If line numbers are to be monitored, then the current value of `yylineno` should be pushed on a stack, and `yylineno` should be reset by `performSwitch()`, while `popSource()` should reinstate a former value of `yylineno` by popping a previous value from the stack. In the current implementation of `Scanner` a simple stack of `yy_buffer_state` pointers is maintained. Changing that into a stack of `pair<yy_buffer_state *, unsigned>` elements allows you to save the line numbers. This modification is left as an exercise to the reader.

As mentioned, `performSwitch()` performs the actual file-switching. The `yyFlexLexer` class provides a series of member functions that can be used for file switching purposes. The file-switching capability of a `yyFlexLexer` object is founded on the `struct yy_buffer_state`, containing the state of the scan-buffer of the file that is currently scanned by the lexical scanner. This buffer is pushed on the state stack when an `#include` is encountered. Then this buffer is replaced by the buffer that will be created for the file that is mentioned in the `#include` directive. The actual file switching is realized as follows:

- First, the current depth of the include-nesting is inspected. If `maxDepth` is reached, the stack is considered full, and the scanner throws a `nestingTooDeep` exception.
- Next, the `fileName` vector is inspected, to avoid circular inclusions. If `nextSource` is encountered in the `fileName` vector, the inclusion is refused, and the scanner throws a `circularInclusion` exception. The member function `throwOnCircularInclusion()` is called for this task.
- Then, a new `ifstream` object is created, for the filename in `nextSource`. If this fails, the scanner throws a `cantRead` exception.
- Finally, a new `yy_buffer_state` is created for the newly opened stream, and the lexical scanner is instructed to switch to that stream using `yyFlexLexer`'s member function `yy_switch_to_buffer()`.

The sources for the member functions `performSwitch()` and `throwOnCircularInclusion()` are:

```

#include "scanner.H"

void Scanner::performSwitch()
{
    if (state.size() == maxDepth)
        throw nestingTooDeep;

    fileName.push_back(nextSource);
    throwOnCircularInclusion();
}

```

```

    ifstream
        *newStream = new ifstream(nextSource.c_str());

    if (!*newStream)
        throw cantRead;

    state.push(yy_current_buffer);
    yy_switch_to_buffer(yy_create_buffer(newStream, sizeof_buffer));
}

```

```

#include "scanner.H"

```

```

void Scanner::throwOnCircularInclusion()
{
    vector<string>::iterator
        it = find(fileName.begin(), fileName.end(), nextSource);

    if (it != fileName.end())
        throw circularInclusion;
}

```

The member function `popSource()` is called to pop the previously pushed buffer from the stack, to continue its scan just beyond the just processed `#include` directive. The `popSource()` function first inspects the size of the state stack: if empty, false is returned and the function terminates. if not empty, then the current buffer is deleted, to be replaced by the state waiting on top of the stack. The file switch is performed by the `yyFlexLexer` members `yy_delete_buffer()` and `yy_switch_to_buffer`. Note that `yy_delete_buffer()` takes care of the closing of the `ifstream` and of deleting the memory allocated for this stream in `performSwitch()`. Furthermore, the filename that was last entered in the `fileName` vector is removed. Having done all this, the function returns true:

```

#include "scanner.H"

```

```

bool Scanner::popSource()
{
    if (state.empty())
        return false;

    yy_delete_buffer(yy_current_buffer);
    yy_switch_to_buffer(state.top());
    state.pop();
    fileName.pop_back();
    return true;
}

```

These functions complete the implementation of the complete lexical scanner. the lexical scanner itself is stored in the `Scanner::yylex()` function. The `Scanner` object itself only has three public member functions and a constructor. One member function is `yylex()`, the other two allow the called to dump a stack trace and to obtain the name of the file that was last processed by the scanner. The constructor is a simple piece of code. Here is its source:

```

#include "scanner.h"

```

```

Scanner::Scanner(istream *yyin, string const &initialName)

```



```

{
    switch_streams(yyin, yyout);
    fileName.push_back(initialName);
}

```

## The main() function

The main program is very simple. As the program expects a filename to start the scanning process at, initially the number of arguments is checked. If at least one argument was given, then a `ifstream` object is created. If this object can be created, then a `Scanner` object is constructed, receiving the address of the `ifstream` object and the name of the file as its arguments. Then the `yylex()` member function of the `Scanner` object is called. This function is inherited from the `Scanner`'s base class `yyFlexLexer`. If anything fails inside the scanner, an exception is thrown, which is caught at the end of the `main()` function:

```

/*                                lexer.cc

    A C++ main()-frame generated by C++ for lexer.cc

*/

#include "lexer.h"                /* program header file */

int main(int argc, char **argv)
{
    if (argc == 1)
    {
        cerr << "Filename argument required\n";
        exit (1);
    }

    ifstream
        yyin(argv[1]);

    if (!yyin)
    {
        cerr << "Can't read " << argv[1] << endl;
        exit(1);
    }

    Scanner
        scanner(&yyin, argv[1]);

    try
    {
        scanner.yylex();
    }
    catch (Scanner::Error err)
    {
        char const *msg[] =
        {
            "Include specification",

```

```

        "Circular Include",
        "Nesting",
        "Read",
    };

    cerr << msg[err] << " error in " << scanner.lastFile() <<
        ", line " << scanner.lineno() << endl;
    scanner.stackTrace();
    return 1;
}
}

```

## Building the scanner-program

The final program is constructed in two steps. These steps are given for a Unix system, on which `flex++` and the Gnu C++ compiler `g++` have been installed:

- First, the lexical scanner's source is created using `flex++`. For this the command

```
flex++ lexer
```

can be given.

- Next, all sources are compiled and linked, using the `libfl.a` library. The appropriate command here is

```
g++ -o scanner *.cc -lfl
```

For the purpose of debugging a lexical scanner the rules that are matched and the tokens that are returned are useful information. When `flex++` is called with the `-d` flag, debugging code will be part of the generated scanner. Apart from that, the debugging code must be activated. Assuming the scanner object is called `scanner`, the statement

```
scanner.set_debug(1);
```

must be given following the construction of the `scanner` object.

### 19.7.2 Using both `bison++` and `flex++`

When an input language exceeds a certain level of complexity, a *parser* is generally needed to be able to control the complexity of the input language. In these cases, a *parser generator* is used to generate the code that is required to determine the grammatical correctness of the input. The function of the lexical scanner is to provide chunks of the input, called *tokens*, for the parser to work with.

Starting point for a program using both a parser and a scanner is the grammar: the grammar is specified first. This results in a set of tokens which can be returned by the lexical scanner (commonly called the *lexer*). Finally, auxiliary code is provided to 'fill in the blanks': the actions which are performed by the parser and the lexer are not normally specified in the grammatical rules or lexical regular expressions, but are executed by functions, which are called from within the parser's rules or associated with the lexer's regular expressions.

In the previous section we've seen an example of a C++ class generated by `flex++`. In the current section we concern ourselves with the parser. The parser can be generated from a grammar specified for the program `bison++`. The specification of `bison++` is similar to the specifications required for `bison`, but a class is generated, rather than a single function. In the next sections we'll develop a program converting *infix expressions*, in which binary operators are written between their operands, to *postfix expressions*, in which binary operators are written behind their operands. A comparable situation holds true for the unary operators `-` and `+`: We ignore the `+` operator, but the `-` will be converted to a unary minus.

Our calculator will recognize a minimal set of operators: multiplication, addition, parentheses, and the unary minus. We'll distinguish real numbers from integers, to illustrate a subtlety in the `bison`-like grammar specifications, but that's about it: the purpose of this section, after all, is to illustrate a C++ program, using a parser and a lexer, and not to construct a full-fledged calculator.

In the next few sections we'll develop the grammar in a `bison++` specification file. Then, the regular expressions for the scanner are specified according to the requirements of `flex++`. Finally the program is constructed.

The class-generating `bison` software (`bison++`) is not widely available. The version used by us is 2.20. It can be obtained at `ftp://ftp.rug.nl/contrib/frank/software/linux/bison/bison++2.20.tar.gz`.

## The `bison++` specification file

The grammar specification file used with `bison++` is comparable to the specification file used with `bison`. Differences are related to the class nature of the resulting parser. Our calculator will distinguish real numbers from integers, and will support the basic set of arithmetic operators.

The `bison++` specification file consists of the following sections:

- The *header section*. This section is comparable to the C specification section used with `bison`. The difference being the `%header{...}` opening. In this section we'll encounter mainly declarations: header files are included, and the `yyFlexLexer` object is declared.
- The *token section*. In this section the `bison` tokens, and the priority rules for the operators are declared. However, `bison++` has several extra items that can be declared here. They are important and warrant a section of their own.
- The *rule section*. The grammatical rules define the grammar. This section has not been changed since the `bison` program.

## The `bison++` token section

The token section contains all the tokens that are used in the grammar, as well as the priority rules as used for the mathematical operators. Moreover, several new items can be declared here:

- `%name ParserName`. The name `ParserName` will be the name of the parser's class. This entry should be the first entry of the token section. It is used in cases where multiple grammars are used, to make sure that the different parser-classes use unique identifiers. By default the name `parse` is used.
- `%define name content`. The `%define` has the same function as the `#define` statement for the C++ preprocessor. It can be used to define, e.g., a macro. Internally, the defined symbol will be the concatenation of `YY_`, the parser's class name, and the name of the macro. E.g.,

`YY_ParserName_name`

Several symbols will normally be defined here. Of all the definitions that can be given here, the following two are *required*:

- `%define LEX_BODY { inline-code }`: here the body of the call to the lexer is defined. It can be defined as `= 0` for an abstract parser-class, but otherwise it must contain the code (including surrounding curly braces) representing the call to the lexer. For example, if the lexer object generated by `flex++` is called `lexer`, this declaration should be

```
%define LEX_BODY { return lexer.yylex(); }
```

- `%define ERROR_BODY { inline-code }`: similarly, the body of the code of the call to the error-function is defined here. It can be defined as `= 0`, in which case the parser's class will again become abstract. Otherwise, it is used to specify the inner workings of the error function, including surrounding braces. E.g.,

```
%define ERROR_BODY { cerr << "syntax Error\n"; }
```

When the `LEX_BODY` and `ERROR_BODY` definitions are omitted, then the compiler is not able to complete the virtual table of the parser class, and the linking phase will report an error like

```
undefined reference to 'Parser virtual table'
```

The remaining symbols are optional, and can be (re)defined as needed:

- `%define DEBUG 1`: if non-0 debugging code will be included in the parser's source.
- `%define ERROR_VERBOSE`: if defined, the parser's stack will be dumped when an error occurs.
- `%define LVAL yylval`: the default variable name is shown here: the variable name containing the parser's semantic value is by default `yylval`, but its name may be redefined here.
- `%define INHERIT : public ClassA, public ClassB`: the inheritance list for the parser's class. Note that it starts with a colon. The space character between `INHERIT` and the colon may be omitted. The `%define` itself should be omitted if the parser class is not derived from another class.
- `%define MEMBERS member-prototypes`: if the parser contains extra members, they must be declared here. Note that there is only one `%define MEMBERS` definition allowed. So, if multiple members are to be declared, they must all be declared at this point. To prevent very long lines in the specification file, the `\`(backslash) can be used at the end of a line, to indicate that it continues on the next line of the source-text. E.g.,

```
%define MEMBERS void lookup(); \
                void lookdown();
```

The `MEMBERS` section starts as a public section. If private members are required too, a `private:` directive can be part of the `MEMBERS` section.

- Constructor-related defines: When a special parser constructor is needed, then three `%defines` can be used:
  - \* `%define CONSTRUCTOR_PARAM parameterlist`: this defines the parameterlist for the parser's constructor. Here the types and names of the parameters of the parser's constructor should be given. The surrounding parentheses of the parameter list are not part of the `CONSTRUCTOR_PARAM` definition.
  - \* `%define CONSTRUCTOR_INIT : initializer(s)`: this defines the base class and member initializers of the constructor. Note the initial colon following `CONSTRUCTOR_INIT`, which is required. The colon may be given immediately after the `CONSTRUCTOR_INIT` statement, or blanks may be used to separate the symbol from the colon.

\* `%define CONSTRUCTOR_CODE { code }`; this defines the code of the parser's constructor (including surrounding curly braces).

When the parser doesn't need special effects, a constructor will not be needed. In those cases the parser can be created as follows (using the default parser-name):

```
parse parser;
```

- `%union`. This starts the definition of the semantical value union. It replaces the `#define YYSTYPE` definition seen with `bison`. An example of a `%union` declaration is

```
%union
{
    int
        i;
    double
        d;
};
```

The 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. As a side line: the lexical scanner does not have to know about this union. The scanner can simply pass its scanned text to the parser through its `YYText()` member function. For example, a statement like:

```
$$i = atoi(scanner.YYText());
```

can be used to convert matched text to a value of an appropriate type.

- Associating tokens and non-terminals with union fields. Tokens and non-terminals can be associated with union fields. This is strongly advised. By doing so, the parser's action implementations become much cleaner than if the tokens aren't associated with fields. As non-terminals can be associated with union fields, the generic return variable `$$` or the generic return values `$1`, `$2`, etc, that are associated with components of rules can be used, rather than `$$i`, `$3.d`, etc. To associate a non-terminal or a token with a union field, the `<fieldname>` specification is used. E.g.,

```
%token <i> INT           // token association (deprecated)
        <d> DOUBLE
%type  <i> intExpr       // non-terminal association
```

In this example, note that both the tokens and the non-terminals can be associated with a field of the union. However, as noted earlier, the lexical 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 knows what the input is all about, and may convert strings like "123" to an integer value. Consequently, we are discouraging the association of a union field and a token. In the upcoming description of the rules of the grammar this will be further illustrated.

- In the `%union` discussion the `%token` and `%type` specifications should be noted. They are used for the specification of the tokens (terminal symbols) that can be returned by the lexical scanner, and for the specification of the return types of non-terminals. Apart from `%token` the token indicators `%left`, `%right` and `%nonassoc` may 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 given 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 discussed in this context. The documentation provided with the `bison` distribution should be consulted for further reference.

## The bison++ grammar rules

The rules and actions of the grammar are specified as usual. The grammar for our little calculator is given below. A lot of rules, but they illustrate the use of non-terminals associated with value-types.

```
lines:
    lines
    line
|
    line
;

line:
    intExpr
    '\n'
    {
        cerr << "int: " << $1 << endl;
    }
|
    doubleExpr
    '\n'
    {
        cerr << "double: " << $1 << endl;
    }
|
    '\n'
    {
        cout << "Good bye\n";
        YYACCEPT;
    }
|
    error
    '\n'
;

intExpr:
    intExpr '*' intExpr
    {
        $$ = $1 * $3;
    }
|
    intExpr '+' intExpr
    {
        $$ = $1 + $3;
    }
|
    '(' intExpr ')'
    {
        $$ = $2;
    }
|
    '-' intExpr          %prec UnaryMinus
    {
        $$ = -$2;
    }
```

```

    }
|
    INT
    {
        $$ = atoi(lexer.YYText());
    }
;

doubleExpr:
    doubleExpr '*' doubleExpr
    {
        $$ = $1 * $3;
    }
|
    doubleExpr '+' doubleExpr
    {
        $$ = $1 + $3;
    }
|
    doubleExpr '*' intExpr
    {
        $$ = $1 * $3;
    }
|
    doubleExpr '+' intExpr
    {
        $$ = $1 + $3;
    }
|
    intExpr '*' doubleExpr
    {
        $$ = $1 * $3;
    }
|
    intExpr '+' doubleExpr
    {
        $$ = $1 + $3;
    }
|
    '(' doubleExpr ')'
    {
        $$ = $2;
    }
|
    '-' doubleExpr          %prec UnaryMinus
    {
        $$ = -$2;
    }
|
    DOUBLE
    {
        $$ = atof(lexer.YYText());
    }
;

```

Using these rules a simple calculator is defined in which integer and real values can be negated, added, and multiplied, and in which standard priority rules can be circumvented using parentheses. The rules show 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 token section of the parser specification file, which is:

```
%name Parser
%union
{
    int i;
    double d;
};
%token      INT
           DOUBLE
%type  <i> intExpr
       <d> doubleExpr

%left  '+'
%left  '*'
%right UnaryMinus

%define MEMBERS \
    virtual ~Parser() {} \
    private: \
        yyFlexLexer lexer;
%define LEX_BODY {return lexer.yylex();}

%define ERROR_BODY { cerr << "error encountered\n"; }
```

In the token section we see the use of the `%type` specifiers, connecting `intExpr` to the `i`-field of the semantic-value union, and connecting `doubleExpr` to the `d`-field. At first sight it looks complex, since the expression rules must be included for each individual return type. On the other hand, if the union itself would have been used, we would have had to specify somewhere in the returned semantic values what field to use: less rules, but more complex and error-prone code.

Also, note that the lexical scanner is included as a member of the parser. There is no need to define the scanner outside of the parser, as it's not used outside of the parser object. The virtual destructor is included as a member to prevent the compiler from complaining about the parser having a non-virtual destructor.

### The flex++ specification file

The flex-specification file that is used by our calculator is simple: blanks are skipped, single characters are returned, and numerical values are returned as either `Parser::INT` or `Parser::DOUBLE` tokens. Here is the complete flex++ specification file:

```
%{
#include "parser.h"
%}
%option yylineno
%%

[ \t]          ;
```



```

[0-9]+                return(Parser::INT);

"."[0-9]*              |
[0-9]+("."[0-9]*)?     return(Parser::DOUBLE);

exit                  |
quit                  return (Parser::DONE);

.|\\n                  return (*yytext);

```

## The generation of the code

The code is generated in the same way as with bison and flex. To order bison++ to generate the files `parser.cc` and `parser.h`, give the command:

```
bison++ -d -o parser.cc parser
```

Next, flex++ will generate code on `lexer.cc` using the command

```
flex++ -I -olexer.cc lexer
```

Note here that flex++ expects no blanks between the `-o` flag and `lexer.cc`.

On Unix systems, linking and compiling the generated sources and the source for the main program (given below) is realized by the command:

```
g++ -o calc -Wall *.cc -lfl -s
```

Note the fact that the `libfl.a` library is mentioned here. If it's not mentioned, the linker will complain about unresolved references like `yywrap()`.

A source in which the `main()` function, the lexical scanner and the parser objects are defined is, finally:

```

#include "parser.h"

int main()
{
    Parser
        parser;

    cout << "Enter (nested) expressions containing ints, doubles, *, + and "
          "unary -\\n"
          "operators. Enter an empty line, exit or quit to exit.\\n";

    parser.yyparse();

    return (0);
}

```

# Index

'0', 57  
-+, 489  
->, 299  
->\*, 299  
-d, 497  
.\*, 299  
.h, 27  
//, 22  
/bin/ls, 477, 480  
/bin/sh, 477, 479, 480  
::, 32, 196, 199  
::delete[], 201  
::new[], 200  
<, 219  
<=, 219  
= 0, 276  
==, 219  
>, 219  
>=, 219  
[&dummy, &dummy), 291  
[begin, end), 222  
[first, beyond), 223, 227, 236, 242, 248  
[first, last), 334  
[left, right), 322  
#define YYSTYPE, 500  
#define \_cplusplus, 25  
#ifdef, 26  
#ifndef, 26  
#include, 15, 489  
#include <algorithm>, 334, 335  
#include <complex>, 257  
#include <deque>, 234  
#include <ext/hash\_map>, 253  
#include <ext/hash\_set>, 257  
#include <filepath>, 489  
#include <fstream>, 80, 89, 96  
#include <functional>, 314  
#include <hashclasses.h>, 254  
#include <iomanip>, 80, 98  
#include <iosfwd>, 76, 79  
#include <iostream>, 27, 79, 87, 93  
#include <istream>, 79, 93  
#include <iterator>, 326–328, 449  
#include <list>, 224  
#include <map>, 237, 244  
#include <memory>, 329  
#include <numeric>, 334  
#include <ostream>, 79, 87  
#include <queue>, 231, 232  
#include <set>, 246, 249  
#include <sstream>, 80  
#include <stack>, 251  
#include <stdio.h>, 22  
#include <streambuf>, 79  
#include <typeinfo>, 285  
#include <utility>, 220  
#include <vector>, 221  
#include directive, 491  
\$\$, 500  
%header{...}, 498  
%option yylineno, 491  
%option yylineo, 493  
\_cplusplus, 25, 26  
\_kernel\_ptrdiff\_t, 484  
0-pointer, 333  
0x30, 57  
  
abort, 177  
abs(), 259  
absolute position, 103, 106  
abstract base class, 292  
abstract class, 468  
abstract classes, 276  
abstract containers, 218, 422  
abstract data types, 313  
access files, 89, 96  
access rules, 442  
access to class members, 211  
access(), 42  
accessor, 214  
accessor member function, 189  
accumulate(), 315, 335  
actions, 491, 497  
adaptors, 313  
add functionality to a template class, 440  
addition, 315, 498  
additional functionality, 269  
address, 212, 417  
address of objects, 156  
adjacent\_difference(), 336

- adjacent\_find(), 337
- adjustfield, 84
- aggregate class, 269
- algorithm, 423, 430
- algorithms, 313
- alias, 212
- allocate arrays, 142
- allocate arrays of objects, 142
- allocate memory, 199
- allocate objects, 141
- allocate primitive types, 141
- allocated memory, 313
- allocation, 146
- allocation error, 151
- allocator class, 313
- alphabetic sorting, 319
- ambiguity, 50, 154, 269, 277–279
- ambiguity: with delete[], 202
- ambiguous, 418, 448
- anachronism, 415
- angular bracket notation, 218, 220, 221, 237
- anonymous, 318, 319, 326, 402, 440
- anonymous complex values, 258
- anonymous namespace, 47
- anonymous object, 161, 193, 195, 206
- anonymous pair, 221
- anonymous string, 61
- anonymous variable: generic form, 221
- ANSI/ISO, 15, 17, 25, 46, 77, 80, 83, 101, 219, 253, 294, 451, 465
- ANSO/ISO, 219
- approach towards iterators, 325
- arg(), 259
- argc, 425
- argument deduction, 415
- argv, 425
- arithmetic function object, 315
- arithmetic operations, 315, 482
- array boundary overflow, 44
- array bounds, 222
- array bounds overflow, 100
- array index notation, 142
- array of objects, 143
- array of pointers to objects, 143
- array-to-pointer conversion, 421
- arrays of fixed size, 142, 143
- arrays of objects, 329
- ASCII, 83, 87, 88, 93, 94, 233
- ASCII collating sequence, 59
- ASCII-Z, 57, 58, 70, 89, 94, 100, 115, 465
- ASCII-Z string, 57
- assembly language, 19
- assert(), 62
- assignment, 159, 270
- assignment operator, 271, 485
- assignment operator: private, 286
- assignment: refused, 271
- assingment: pointers to members, 298
- associative array, 237, 244, 253
- associativity, 157
- associativity of operators, 500
- atoi(), 97
- auto-assignment, 156
- auto\_ptr, 313, 329, 465
- auto\_ptr: 0-pointer, 332
- auto\_ptr: assigning new content, 333
- auto\_ptr: assignment, 332
- auto\_ptr: defining, 330
- auto\_ptr: empty, 332
- auto\_ptr: initialization, 330, 331
- auto\_ptr: losing its memory, 332
- auto\_ptr: operators, 333
- auto\_ptr: reaching members, 331
- auto\_ptr: restrictions, 329
- auto\_ptr: used type, 331
- auto\_ptr<>::get(), 333
- auto\_ptr<>::operator\*(), 333
- auto\_ptr<>::operator->(), 333
- auto\_ptr<>::operator=(), 333
- auto\_ptr<>::release(), 333
- auto\_ptr<>::reset(), 333
- automatic expansion, 222
- available member functions, 272
- back\_inserter(), 325
- background process, 481
- backslash, 499
- bad\_cast, 285
- base class, 260, 262, 264, 265, 267, 268, 271, 273, 276, 277, 279, 283, 312, 431, 439, 460, 490
- base class constructor, 280
- base class destructor, 265
- base class initializer, 264, 499
- base class initializer: ignored, 280
- base class initializers: calling order, 269
- base class pointer, 273, 275
- base class: converting to derived class, 283
- base class: coverting to derived class, 283
- base class: instantiation, 423
- base classes: merged, 280
- bash, 109
- BASIC, 19
- basic data types, 41
- basic exception handling, 176
- basic operators of containers, 219
- basic\_, 76
- basic\_ios.h, 80
- begin(), 322

- bidirectional\_iterator, 449
- BidirectionalIterator, 450
- BidirectionalIterators, 324, 449
- binary and, 86
- binary file, 91, 113
- binary files, 88, 94, 112
- binary function object, 321
- binary function objects, 321
- binary input, 93
- binary operator, 318, 482
- binary or, 86
- binary output, 83, 87
- binary tree, 410
- binary\_search(), 338
- bind1st(), 321
- bind2nd(), 321
- binder, 321
- bison, 488, 498
- bison++, 498
- bison++: <fieldname>, 500
- bison++: %define, 498
- bison++: %define CONSTRUCTOR\_CODE, 500
- bison++: %define CONSTRUCTOR\_INIT, 499
- bison++: %define CONSTRUCTOR\_PARAM, 499
- bison++: %define DEBUG, 499
- bison++: %define ERROR\_BODY, 499
- bison++: %define ERROR\_VERBOSE, 499
- bison++: %define INHERIT, 499
- bison++: %define LEX\_BODY, 499
- bison++: %define LVAL, 499
- bison++: %define MEMBERS, 499
- bison++: %define: required, 499
- bison++: %left, 500
- bison++: %name, 498
- bison++: %nonassoc, 500
- bison++: %prec, 500
- bison++: %right, 500
- bison++: %token, 500
- bison++: %type, 500
- bison++: %union, 500
- bison++: associating token and union field, 500
- bison++: code generation, 504
- bison++: obtaining, 498
- bison++: using YYText(), 500
- bitfunctional, 482
- bitwise, 482
- bitwise and, 82, 482
- bitwise operations, 315, 482
- bookkeeping, 329
- bool, 41, 241, 248
- boolean operators, 82
- bootstrapping problem, 202
- bound friend, 436, 440, 484
- bound friend template, 435
- bound template function, 437
- boundary overflow, 184
- buffer, 77, 103, 106, 455
- building blocks, 269
- bytewise comparison, 203
- bytewise copy, 151
- C++ library, 219
- calculator, 498, 503
- calculators, 251
- call overloaded operators, 155
- callable member functions, 275
- calling order of base class initializers, 269
- calloc(), 140
- candidate functions, 428
- capsbuf, 289
- case insensitive comparison of strings, 60
- case sensitive, 314
- cast, 190
- catch, 166, 171, 177, 312
- catch all expressions, 178
- catch: all exceptions, 179
- categories of generic algorithms, 334
- cerr, 33, 87, 108, 187
- chain of command, 101
- Chain of Responsibility, 468, 472
- char, 76
- char \*, 188
- char const \*, 253
- Character set searches, 64
- characteristics of iterators, 449
- chardupnew(), 140
- cheating, 269
- child process, 467
- ChildPlain, 472, 477
- ChildProcess, 468
- cin, 33, 79, 93
- circular inclusions, 494
- class, 46, 312, 415
- class derivation, 439
- class hierarchies, 422
- class hierarchy, 260, 274
- class implementation, 119
- class interface, 119, 263, 266, 276, 431, 435
- class name, 286
- class vs. typename, 415
- class: abstract, 276
- classes: derived from streambuf, 455
- classes: having non-pointer data, 163
- classes: without data members, 277
- clear(), 112
- Cline, 36
- clog, 87
- closed namespace, 47

- closing streams, 90, 96
- code generation, 504
- collision, 253
- colon, 499
- combined reading and writing using streams, 79
- Command design pattern, 469–471
- command-line, 489
- comment-lines, 489
- common data fields, 208
- common pool, 458
- common practice, 491
- comparator, 319
- compilation error, 302
- compilation time, 425
- compile-time, 273, 281, 294
- compiler, 16, 18, 276, 434
- complex, 257
- complex container, 219
- complex numbers, 219, 257
- complex::operator\*(), 259
- complex::operator+(), 258
- complex::operator+=(), 259
- complex::operator-(), 258
- complex::operator-=(), 259
- complex::operator/(), 259
- composed const object, 132
- composition, 131, 136, 260, 267, 433
- compound statement, 177
- concatenated assignment, 158
- concatenation of closing angular brackets, 238
- condition flags, 81
- condition member functions, 82
- condition state, 81
- conj(), 259
- const, 34, 422
- const &, 185
- const data and containers, 220
- const data member initialization, 132
- const functions, 36
- const member functions, 122, 127, 277
- const objects, 161, 195, 431
- const\_cast<type>(expression), 24
- const\_iterator, 449
- constant expression, 434
- constant expression, 416
- constant function object, 321
- constructing pointers, 297
- construction: template class, 431
- constructor, 103, 120, 140, 199, 206, 264, 269, 313, 320, 327, 328, 431, 493
- constructor characteristics, 159
- constructor: as template, 443
- constructor: calling order, 265
- constructor: implicit use, 193
- constructor: primary function, 121
- constructor: private, 164
- constructors having one parameter, 191
- constructors: and unions, 500
- container without angular brackets, 220
- containers, 218, 313
- containers storing pointers, 220
- containers: basic operators, 219
- containers: data type requirements, 219
- containers: equality tests, 219
- containers: initialization, 222
- containers: nested, 238
- containers: ordering, 219
- containers and const data, 220
- contrary to intuition, 238
- conversion, 424
- conversion operator, 190, 276
- conversion operator: with insertions, 191
- conversion rules, 41
- conversion to a base class, 423, 424
- conversions, 92, 97, 421, 434
- copy constructor, 159, 162, 164, 195, 222, 226, 231, 234, 235, 239, 247, 252, 263, 332, 353, 440, 485
- copy constructor: double call, 195
- copy constructor: private, 286
- copy files, 107
- copy information, 491
- copy non-involved data, 225
- copy objects, 153
- copy(), 162, 163, 339, 437
- copy\_backward(), 340
- cos(), 259
- cosh(), 259
- count(), 341
- count\_if(), 321, 342
- cout, 33, 79, 87, 108, 187, 466
- create files, 89
- create values, 237, 247
- cstdint, 196
- cstdlib, 448
- Cygnus, 18
- Cygwin, 18
- daemon, 468, 481
- data base, 113
- data hiding, 19, 43, 211, 213, 215, 263
- data integrity, 213
- data members, 103, 263
- data members: multiply included, 281
- data organization, 195
- data structure, 430
- data structures, 313, 434
- data type, 253, 430

- data.cc, 209
- database applications, 88, 95
- deallocate memory, 199
- Debian, 18
- debugging, 497
- decimal format, 98
- declaration, 420, 425–427, 430, 432, 445
- declarations, 498
- declarative region, 46
- declare iostream classes, 76
- decrement operator, 193, 450
- default, 199, 490, 498, 499
- default arguments, 29, 432
- default constructor, 121, 131, 142, 159, 199, 219, 222, 264, 314, 326
- default copy constructor, 162
- default exception handler, 179
- default implementation, 105
- default initialization, 125
- default operator delete, 199
- default parameter values, 125, 191
- default value, 192, 222, 223, 228, 236
- define members of namespaces, 54
- definition, 430, 445
- definition: template member function, 435
- definitions of static members, 438
- delete, 140, 141, 198, 275, 333
- delete[], 142, 145, 147
- delete[]: ignored, 148
- deletions, 224
- delimiter, 327
- dependencies between code and data, 260
- deprecated, 415
- deque, 234, 322, 324
- deque constructors, 235
- deque::back(), 235
- deque::begin(), 235
- deque::clear(), 235
- deque::empty(), 235
- deque::end(), 235
- deque::erase(), 235
- deque::front(), 236
- deque::insert(), 236
- deque::pop\_back(), 236
- deque::pop\_front(), 236
- deque::push\_back(), 236
- deque::pushfront(), 236
- deque::rbegin(), 236
- deque::rend(), 236
- deque::resize(), 236
- deque::size(), 236
- deque::swap(), 236
- dereference, 299, 333, 449
- dereferencing, 299
- derivation, 260, 262
- derived class, 260, 264, 268, 271, 273, 276, 277, 279, 283, 431, 465, 490
- derived class destructor, 265
- derived template class, 439
- dervied class, 312
- design considerations, 434
- design pattern, 468, 469, 472
- design patterns, 467
- destroy(), 162
- destructor, 120, 144, 199, 263, 265, 275, 353, 431, 452, 485
- destructor: calling order, 265
- destructor: empty, 276
- destructor: inline, 276
- destructor: not required, 485
- destructor: when to define, 275
- device, 79, 81, 101, 106, 108, 288, 451
- direct base class, 263
- direct member, 263
- directive, 499
- disambiguate, 191
- disambiguation, 427
- disastrous event, 168
- divides<>(), 318
- division, 315
- division by zero, 175
- DOS, 112
- doubly ended queue data structure, 234
- downcasts, 285
- dup(), 466
- dup2(), 466, 473, 474
- duplication of data members, 281
- dynamic allocation, 465
- dynamic arrays, 142, 143
- dynamic binding, 273
- dynamic cast, 283
- dynamic cast: prerequisite, 283
- dynamic growth, 225
- dynamic\_cast<>(), 25, 283, 285, 292, 312
- dynamically allocated, 333
- dynamically allocated memory, 263, 329
- dynamically allocated variables, 434
- early binding, 273
- ECHO, 491
- efficiency, 253
- empty, 322
- empty deque, 236
- empty destructor, 276
- empty enum, 312
- empty function throw list, 180
- empty list, 228
- empty parameter list, 25
- empty strings, 65

- empty throw, 179
- empty vector, 223
- encapsulation, 103, 196, 213, 215, 216
- end of line comment, 22
- end(), 322
- end-of-stream, 326, 327
- endl, 34
- enlarge an array, 142, 143
- enum, 30
- enumeration: nested, 310
- enumerations: nested, 441
- equal(), 343
- equal\_range(), 344
- equal\_to<>(), 319
- equality operator, 219
- error code, 165
- ERROR\_BODY, 499
- escape mechanism, 213
- exception, 83, 168, 171, 284, 494
- exception handler, 175, 312
- exception handler: order, 178
- exception: cases, 178
- exception: construction of, 179
- exception: default handling, 177
- exception: dynamically generated, 177
- exception: levels, 177
- exception: outside of try block, 177
- exceptions, 165
- exceptions: when, 174
- exercise, 118, 200, 429, 465, 482, 494
- exit(), 144, 165
- exp(), 259
- expandable array, 221
- explicit, 193
- explicit argument list, 202
- explicit argument specification, 426
- explicit argument type list, 424
- explicit arguments, 198
- explicit construction, 193
- explicit insertion, 237, 247
- explicit instantiation declaration, 424
- explicit return, 23
- explicit specialization definition, 445
- explicit template argument, 428
- explicit template class specialization, 446
- explicit template instantiation declaration, 425
- expression, 251
- expression: actual type, 283, 286
- extendable array, 218
- extern, 433
- extra blank space, 258
- extracting information from memory buffers, 79
- extracting strings, 94
- extraction manipulators, 100
- extraction operator, 33, 93, 187
- failure, 106
- failure::what(), 182
- false, 41, 68, 361, 371, 491
- field selector, 299
- field selector operator, 31
- field width, 205
- FIFO, 218, 231
- FILE, 76
- file descriptor, 90, 109, 451, 459, 460, 462
- file descriptors, 79, 451, 473
- file flags, 90
- file is rewritten, 91
- file modes, 90
- file stack, 493
- file switch, 494
- file switching, 494
- filebuf, 79, 90, 106
- filebuf::close(), 107
- filebuf::filebuf(), 106
- filebuf::is\_open(), 107
- filebuf::open(), 107
- fill characters, 84
- fill(), 346
- fill\_n(), 346
- FIFO, 219, 251
- find(), 347
- find\_end(), 349
- find\_first\_of(), 351
- find\_if(), 348
- first, 220, 238
- first data member, 294
- first in, first out, 218, 231
- first in, last out, 218, 251
- fixed size arrays, 142
- fixed size arrays, 143
- flex, 488, 493
- flex++, 489, 491, 492, 497, 498, 504
- FlexLexer.h, 493
- flow-breaking situations, 165
- flushing a stream, 99
- fool the compiler, 50
- fopen(), 87, 93
- for\_each(), 352
- fork(), 15, 451, 467, 481
- form(), 88, 465
- formal name, 416
- format flags, 98
- formatted input, 93
- formatted output, 83, 87
- formatting, 80, 85, 465
- formatting flags, 83
- forward class reference, 136

- forward declaration, 308, 309
- forward declarations, 76, 306
- forward\_iterator, 449
- ForwardIterator, 450
- ForwardIterators, 324, 449
- fprintf(), 77
- free compiler, 18
- Free Software Foundation, 18
- free(), 140, 146
- freeze(0), 92
- friend, 206, 213–215, 306, 435, 437
- friend declarations, 215, 435
- friend function: synonym for a member, 216
- friendship among classes, 213
- front\_inserter(), 325
- FSF, 18
- fstream, 111
- fstream: and cin, cout, 89
- fstream: reading and writing, 111
- ftp.rug.nl:/.../bison, 498
- ftp://ftp.rug.nl:/.../bison/, 489
- ftp://ftp.rug.nl:/.../cplusplus.annotations, 1
- ftp://ftp.rug.nl:/.../icmake, 19
- ftp://research.att.com/dist/c++std/WP/, 15
- ftp://rzbsdi01.uni-trier.de:/.../bison++/, 489
- ftp://prep.ai.mit.edu/pub/gnu, 489
- fully qualified names, 54
- function adaptors, 314, 320
- function address, 16, 212
- function call operator, 202, 253, 314
- function object, 253, 313
- function objects, 202, 313
- function overloading, 28
- function prototype, 266
- function selection, 428
- function selection: ordinary vs. template, 429
- function throw list, 180
- function-to-pointer conversion, 422
- function: address, 294
- functionality, 221
- functions as part of a struct, 31
- functions having identical names, 28, 31
- functions: ordering of, 428
- g++, 15, 16, 18, 22, 92, 253, 434, 497
- Gamma, E., 467–470
- general purpose library, 313
- general rule, 271, 447
- generate(), 354
- generate\_n(), 355
- generic algorithm, 203, 302, 313, 437, 482
- generic algorithms, 15, 202, 219, 313, 334, 449
- generic data type, 334
- generic return, 500
- generic software, 76
- generic type, 220
- global, 302
- global function, 211, 302
- global object, 120
- global operator delete[], 201
- global operator new[], 200
- global scope, 296, 299
- global variable, 434
- global variables, 208, 251
- Gnu, 15, 16, 18, 22, 92, 150, 253, 310, 434, 451, 465, 497
- goto, 165
- grammar, 497
- grammar specification file, 498
- grammatical correctness, 497
- grammatical rules, 497, 498
- greater<>(), 313, 319
- greater\_equal<>(), 319
- greatest common denominator, 261
- hash function, 253
- hash value, 253
- hash\_map, 253
- hash\_multimap, 253
- hash\_multiset, 253
- hash\_set, 253
- hashclasses.h, 254
- hashing, 253
- hashtable, 219
- header file, 138, 254, 445, 493
- header file: organization, 134
- header files, 46, 79
- header section, 498
- heap, 410
- hex, 205, 290
- hexadecimal, 205
- hexadecimal format, 99
- hidden constructor call, 195
- hidden data member, 294
- hidden object, 161
- hidden pointer, 197
- hiding member functions, 266
- hierarchy of code, 260
- html, 15
- http://gcc.gnu.org, 18
- http://sources.redhat.com, 18
- http://www.cplusplus.com/ref, 17
- http://www.cygnum.com/.../dec96pub, 17
- http://www.debian.org, 18
- http://www.gnu.org, 18
- http://www.linux.org, 18
- http://www.ora.com/.../lex/noframes.html, 488
- http://www.research.att.com/..., 35
- http://www.sgi.com/.../STL, 219
- http://www.xs4all.nl/.../yodl/, 14



<http://www.parashift.com/c++-faq-lite/>, 36  
 human-readable, 83  
 hyperlinks, 17  
  
 I/O, 76, 182  
 I/O library, 76  
 icmake, 19  
 identically named member functions, 269  
 identifier rules, 416  
 ifdnstreambuf, 456  
 ifdseek, 459  
 ifdstreambuf, 455, 459  
 ifstream, 93, 96, 112, 118  
 ifstream constructors, 96  
 ifstream::close(), 96  
 ifstream::open(), 96  
 imaginary part, 257, 259  
 implement an iterator, 449  
 implementation, 209, 276  
 implementation dependent, 213  
 implicit argument, 198  
 implicit conversion, 192, 271  
 import all the names, 49  
 INCLUDE, 135, 137  
 INCLUDE path, 254  
 includes(), 355  
 increment operator, 193, 449  
 index operator, 184, 214, 222, 235, 239, 244  
 indices: vs. numbers, 441  
 indirect base class, 263  
 inequality operator, 219  
 infix expressions, 498  
 inheritance, 260, 262, 264, 490  
 inheritance list, 499  
 init, 468, 481  
 initialization, 141, 159, 222  
 initialization of objects, 127  
 initialization: static data member, 209  
 initialize memory, 140  
 inline, 128, 202–204, 212, 216, 266, 276, 314, 435, 443  
 inline code, 129  
 inline function, 129  
 inline function: placement, 130  
 inline in the function definition, 129  
 inline member functions, 305  
 inline static member functions, 212  
 inline: disadvantage, 130  
 inner\_product(), 357  
 inplace\_merge(), 359  
 input, 93, 101, 103, 108  
 input language, 497  
 input mode, 96  
 input operations, 288, 327  
 input-language, 489  
 input\_iterator, 449  
 InputIterator, 449  
 InputIterator1, 324  
 InputIterator2, 324  
 InputIterators, 324, 449  
 insert formatting commands, 77  
 insert information in memory buffers, 77  
 insert streambuf \*, 108  
 insert(), 325  
 inserter, 325  
 inserter(), 325  
 insertion operator, 33, 77, 79, 87, 187, 213, 435  
 insertion operator: with conversions, 191  
 insertion sequence, 206  
 insertions, 224  
 instantiated, 218  
 instantiation, 254, 313, 414, 419, 438, 442  
 int main(), 23  
 integral conversions, 434  
 interface, 209, 489, 493  
 interface functions, 122  
 intermediate class, 282  
 internal buffer, 89  
 internal organization, 263  
 Internet, 17  
 IORedirector, 472, 475  
 ios, 77, 80, 101, 108, 111, 310, 460  
 ios::adjustfield, 83, 86  
 ios::app, 90, 111, 112  
 ios::ate, 91, 112  
 ios::bad(), 82  
 ios::badbit, 81  
 ios::basefield, 83, 84, 86  
 ios::beg, 88, 96, 310  
 ios::bin, 112  
 ios::binary, 91  
 ios::boolalpha, 84, 98  
 ios::clear(), 82  
 ios::copyfmt(), 85  
 ios::cur, 88, 96, 310  
 ios::dec, 84, 86, 98  
 ios::end, 89, 96  
 ios::endl, 99  
 ios::ends, 99  
 ios::eof(), 82  
 ios::eofbit, 81  
 ios::exception, 182  
 ios::exceptions(), 182  
 ios::fail, 89, 90, 96  
 ios::fail(), 82  
 ios::failbit, 81  
 ios::failure, 182  
 ios::fill(), 85

- ios::fixed, 84, 86, 99
- ios::flags(), 86
- ios::floatfield, 84, 86
- ios::flush, 99
- ios::good(), 82
- ios::goodbit, 81
- ios::hex, 84, 86, 99
- ios::in, 91, 96, 111
- ios::internal, 84, 86, 99
- ios::left, 84, 86, 99
- ios::noboolalpha, 84, 99
- ios::noshowbase, 84, 99
- ios::noshowpoint, 99
- ios::noshowpos, 99
- ios::noskipws, 99
- ios::nounitbuf, 99
- ios::nouppercase, 99
- ios::oct, 84, 86, 99
- ios::openmode, 90, 107, 291
- ios::out, 89, 91, 111
- ios::precision(), 86
- ios::rdbuf(), 81, 109, 466
- ios::rdstate(), 82
- ios::resetiosflags, 86
- ios::resetiosflags(), 100
- ios::right, 84, 86, 100
- ios::scientific, 84, 86, 100
- ios::seekdir, 88, 96, 103
- ios::setbase(), 100
- ios::setf(), 86
- ios::setf(fmtflags flags), 86
- ios::setfill(), 85, 100
- ios::setiosflags, 86
- ios::setiosflags(), 100
- ios::setprecision(), 86, 100
- ios::setstate(), 83
- ios::setstate(int flags), 83
- ios::setw(), 86, 100
- ios::showbase, 84, 100
- ios::showpoint, 85, 100
- ios::showpos, 85, 100
- ios::skipws, 85, 101, 327
- ios::tie(), 81
- ios::trunc, 91
- ios::unitbuf, 85, 89, 101
- ios::unsetf(), 86
- ios::uppercase, 85, 101
- ios::width(), 86
- ios::ws, 101
- ios\_base, 77, 80
- ios\_base.h, 80
- ios\_base::ios\_base(), 80
- iostate, 182
- iostream, 33, 187, 326, 328

- IPipe, 474
- is\_open, 90, 97
- iscanstream, 94, 465
- istream, 79, 93, 118, 187, 287, 325, 327, 455, 491
- istream constructor, 93
- istream::gcount(), 94
- istream::get(), 94
- istream::getline(), 94
- istream::ignore(), 95
- istream::peek(), 95
- istream::putback(), 95, 455, 461
- istream::read(), 95
- istream::readsome(), 95
- istream::seekg(), 95
- istream::tellg(), 95
- istream::unget(), 95, 461
- istream::ungetc(), 455
- istream\_iterator, 327
- istream\_iterator<Type>(), 325
- istreambuf\_iterator, 327, 328
- istreambuf\_iterator<>(), 327
- istreambuf\_iterator<Type>(istream), 327
- istreambuf\_iterator<Type>(streambuf \*), 327
- istreamstream, 79, 93, 97
- istreamstream constructors, 97
- istreamstream::str(), 97
- iter\_swap(), 360
- iterator, 222, 226, 235, 240, 248, 303, 322
- iterator class, 449
- iterator range, 223, 227, 236, 242, 248
- iterators, 219, 220, 222, 313, 431, 449
- iterators: characteristics, 324
- iterators: general characteristics, 322
- iterators: pointers as, 324
- Java, 283
- key, 237
- key type, 253
- key/value, 237
- keyword, 415
- kludge, 194, 292
- Koenig lookup, 49
- Lakos, J., 137
- late binding, 273–275
- late bining, 273
- lazy mood, 138
- left-hand, 219
- left-hand value, 184
- leftover, 381, 406
- legibility, 237, 247
- less-than operator, 219
- less<>(), 319

- less\_equal<>(), 319
- lex, 488
- LEX\_BODY, 499
- lexer, 497, 499
- lexical scanner, 489, 491, 497, 500
- lexical scanner specification, 490, 491
- lexicographic comparison, 69
- lexicographical ordering, 59
- lexicographical\_compare(), 361
- libfl.a, 497, 504
- library, 130, 138
- lifetime, 251, 452
- LIFO, 219, 251
- line number, 493
- line numbers, 491
- linear search, 202
- linear derivation, 267
- linear search, 204
- lineno(), 493
- linker, 276, 504
- Linux, 18, 19, 484
- Lisp, 19
- list, 218, 224, 324
- list constructors, 226
- list data structure, 224
- list traversal, 224
- list::back(), 226
- list::begin(), 226
- list::clear(), 226
- list::empty(), 226
- list::end(), 226
- list::erase(), 226
- list::front(), 227
- list::insert(), 227
- list::merge(), 227
- list::pop\_back(), 228
- list::pop\_front(), 228
- list::push\_back(), 228
- list::push\_front(), 228
- list::rbegin(), 228
- list::remove(), 228
- list::rend(), 229
- list::resize(), 228
- list::reverse(), 229
- list::size(), 229
- list::sort(), 229
- list::splice(), 229
- list::swap(), 230
- list::unique(), 230
- local arrays, 142
- local object, 120
- local program development, 415
- local variable, 416
- local variables, 27, 251, 434
- location of throw statements, 175
- log(), 259
- logical function object, 320
- logical operations, 320, 482
- logical operators, 320
- logical\_and<>(), 320
- logical\_not<>(), 320
- logical\_or<>(), 320
- longjmp(), 165, 168, 172
- longjmp(): alternative to, 170
- longjmp(): avoid, 170
- lower\_bound(), 363
- lsearch(), 202
- lseek(), 460
- Ludlum, 49
- lvalue, 184, 194, 324, 332, 333, 486
- lvalue transformation, 424
- lvalue transformations, 421, 434
- lvalue-to-rvalue conversion, 421
- macro, 206, 498
- main(), 23, 425, 489
- make, 19
- make\_heap(), 411
- malloc(), 140, 146, 151
- manipulator, 205
- manipulators, 77, 85, 98
- manipulators requiring arguments, 206
- map, 219, 237
- map constructors, 237
- map: member functions, 240
- map::begin(), 240
- map::clear(), 240
- map::count(), 240, 244
- map::empty(), 240
- map::end(), 240
- map::equal\_range(), 240
- map::erase(), 240
- map::find(), 240
- map::insert(), 241
- map::lower\_bound(), 242
- map::rbegin(), 242
- map::rend(), 242
- map::size(), 242
- map::swap(), 242
- map::upper\_bound(), 242
- Marshall Cline, 36
- mask value, 83
- matched text, 492, 500
- matched text length, 493
- mathematical functions, 259
- max heap, 410
- max(), 364
- max-heap, 334, 411
- max\_element(), 365

- member function, 57, 273, 433
- member function: called explicitly, 266
- member functions, 40, 103, 180, 215, 226, 231, 234, 235, 252, 333, 434
- member functions: available, 272
- member functions: callable, 275
- member functions: hidden, 266
- member functions: identically named, 269
- member functions: not implemented, 164
- member functions: omitting, 164
- member functions: preventing their use, 164
- member functions: redefining, 265
- member initialization, 131
- member initialization order, 132
- member initializer, 163, 499
- member initializers, 434
- member template, 442
- member: class as member, 303
- members of nested classes, 304
- memory allocation, 140
- memory consumption, 294
- memory leak, 92, 145, 148, 164, 174, 177, 200, 220, 275, 329
- memory leaks, 140
- memory: automatically deleted, 329
- merge(), 366
- merging, 335
- methods, 40
- min(), 368
- min\_element(), 369
- mini scanner, 491
- minus<>(), 318
- missing predefined function objects, 482
- mixing C and C++ I/O, 79
- modifier, 214
- modifiers, 188
- modulus, 315
- modulus<>(), 318
- MS-DOS, 91, 112
- MS-WINDOWS, 112
- MS-Windows, 18, 91
- multimap, 244
- multimap: member functions, 244
- multimap: no operator[], 244
- multimap::equal\_range(), 245
- multimap::erase(), 244
- multimap::find(), 245
- multimap::insert(), 245
- multimap::iterator, 245
- multimap::lower\_bound(), 245
- multimap::upper\_bound(), 245
- multiple derivation, 267, 268
- multiple grammars, 498
- multiple inclusions, 26
- multiple inheritance, 267, 277
- multiple inheritance: which constructors, 280
- multiple parent classes, 267
- multiple virtual base classes, 280
- multiplication, 315, 498
- multiplication operator, 417
- multiplies<>(), 318
- multiset, 249
- multiset: member functions, 249
- multiset::equal\_range(), 249
- multiset::erase(), 249
- multiset::find(), 250
- multiset::insert(), 250
- multiset::iterator, 250
- multiset::lower\_bound(), 250
- multiset::upper\_bound(), 250
- name collisions, 135
- name conflicts, 21
- name lookup, 27
- name mangling, 28
- name resolution, 448
- names of people, 237
- namespace, 21, 138
- namespace alias, 54
- namespace declarations, 47
- namespaces, 46
- nav-com set, 267
- needless code, 261
- negate<>(), 318
- negation, 315
- negators, 321
- nested class, 303, 440, 484
- nested class members: access to, 308
- nested classes: declaring, 306
- nested classes: having static members, 305
- nested containers, 238
- nested derivation, 263
- nested enumerations, 310
- nested inheritance, 277
- nested namespace, 52
- nested template class, 440, 442
- nested template function, 442
- nesting depth, 489
- new, 140, 141, 196
- new-style casts, 23
- new.h, 150
- new\_handler, 140
- new[], 142, 143, 196
- next\_permutation(), 371
- no buffering, 106
- no data members, 277
- no destructor, 148
- non-constant member functions, 277
- non-existing variables, 175

- non-local exit, 165
- non-local exits, 165
- non-static member functions, 197
- non-template function, 426
- non-terminals, 500
- non-type parameter, 416
- nontemplate friend function, 435
- norm(), 259
- not1(), 321
- not2(), 321
- not\_equal\_to<>(), 319
- notation, 142
- notational convention, 220
- nstantiation, 445
- nth\_element(), 373
- NULL, 22, 140
- null-bytes, 89
- numbers: vs. indices, 441
- object, 31, 120
- object address, 156
- object as argument, 160
- object duplication, 153
- object hierarchy, 260
- object oriented approach, 20
- object oriented programming, 431
- object return values, 161
- object-oriented, 260
- objects as data members, 131
- obsolete binding, 27
- octal format, 99
- off\_type, 88, 95
- ofstream, 465
- ofstream, 87, 89, 112, 118
- ofstream constructors, 89
- ofstream::close(), 90
- ofstream::open(), 90
- old-style casts, 23
- omit member functions, 164
- openmode, 91
- operating system, 467, 484
- operator, 155
  - (), 320
  - =(), 68, 203, 319
- operator delete, 198, 199
- operator delete[], 199, 201
- operator new, 142, 196, 199, 330
- operator new[], 142, 199
- operator overloading, 154, 184
- operator overloading: within classes only, 207
- operator string(), 276
- operator(), 202, 204, 253, 314, 380
- operator\*(), 318, 324, 449, 450
- operator+(), 316, 318, 335, 450
- operator++(), 193, 324, 449, 450
- operator-(), 318, 450
- operator-(), 193, 450
- operator/(), 318
- operator<(), 253, 319, 365, 366, 371, 374, 375, 378, 393, 395–397, 399, 400, 409, 411, 412
- operator<<(), 87, 187, 259, 327, 402, 430
- operator<<(): and manipulators, 206
- operator<=(), 319
- operator=(), 431
- operator==(), 68, 319, 324, 391, 392, 406, 407, 449
- operator>(), 314, 319
- operator>=(), 319
- operator>>(), 93, 187, 188, 238, 259, 325
- operator%(), 318
- operator&(), 482
- operator&&(), 320
- operator^(), 482
- operator|(), 86
- operator||(), 320
- operators: associativity, 500
- operators: of containers, 219
- operators: precedence, 500
- operators: priority, 500
- operator[], 189
- operator[](), 184, 244, 431
- OPipe, 474
- options, 491
- ordered pair, 259
- ostream, 77, 79, 81, 87, 118, 187, 205, 206, 276, 287, 290, 327, 328, 402, 435, 444, 465, 491
- ostream constructor, 87
- ostream coupling, 108
- ostream::flush(), 89
- ostream::put(), 88
- ostream::seekp(), 88
- ostream::tellp(), 88
- ostream::write(), 88
- ostream\_iterator, 328
- ostream\_iterator<>(), 437
- ostream\_iterator<Type>(), 327
- ostreambuf\_iterator, 327, 328
- ostreambuf\_iterator<>(), 328
- ostreambuf\_iterator<Type>(streambuf \*), 328
- ostreamstringstream, 77, 87, 91
- ostreamstringstream constructors, 91
- ostreamstringstream::str(), 91
- ostrstream, 92
- out of scope, 275, 329, 331, 452, 458, 485
- output, 87, 102, 105, 108
- output formatting, 77, 80
- output mode, 89

- output operations, 288, 328, 451
- output\_iterator, 449
- OutputIterator, 450
- OutputIterators, 324, 449
- overloadable operators, 207
- overloaded assignment, 158, 159, 162–164, 184, 219
- overloaded assignment operator, 157, 263
- overloaded extraction operator, 188
- overloaded global operator, 187
- overloaded increment: called as operator++(), 196
- overloaded operator, 198
- overloading: template functions, 426
- overview of generic algorithms, 219
- pair, 238
- pair container, 218, 220
- pair<map::iterator, bool>, 241
- pair<set::iterator, bool>, 248
- pair<type1, type2>, 221
- parameter list, 28, 201, 416, 499
- parameters, 415
- parameters: of template functions, 425
- parent, 263
- parent process, 467
- ParentCmd, 472, 477
- parentheses, 498, 499
- ParentProcess, 469
- ParentSlurp, 472, 479
- parse, 498
- parser, 488, 491, 497
- parser generator, 488, 497
- parser: members, 499
- partial specialization, 446
- partial\_sort(), 374
- partial\_sort\_copy(), 375
- partial\_sum(), 376
- partition(), 377
- pdf, 1, 15
- peculiar syntax, 204
- penalty, 274
- permuting, 335
- phone book, 237
- Pipe, 473
- pipe, 451, 473
- pipe(), 473
- plus<>(), 315, 318
- point of instantiation, 430
- pointed arrows, 258
- pointed brackets, 426
- pointer arithmetic, 450, 485, 487
- pointer data, 164
- pointer in disguise, 272
- pointer juggling, 225
- pointer notation, 297
- pointer to a function, 206, 419
- pointer to a pointer, 148
- pointer to an object, 272
- pointer to function, 211
- pointer to function members: using (), 300
- pointer to member, 302
- pointer to member field selector, 299
- pointer to member: access within a class, 301
- pointer to members, 296
- pointer to members: defining, 297
- pointer to objects, 439
- pointer: to template class, 433
- pointers, 322
- pointers to data members, 302
- pointers to deleted memory, 152
- pointers to functions, 202, 203
- pointers to member, 16
- pointers to members: assignment, 298
- pointers to objects, 199
- pointers: as iterators, 324
- polar(), 259
- polymorphism, 25, 176, 273, 292, 294, 431
- pop\_heap(), 411
- pos\_type, 88, 95
- posix\_types.h, 484
- postfix, 450
- postfix expressions, 498
- postfix operator, 193
- postponing decisions, 165
- PostScript, 15
- pow(), 259
- preamble, 491
- precedence of operators, 500
- precompiled template classes, 434
- predefined function objects, 205, 314, 482
- predefined function objects: missing, 482
- predicate, 321
- prefix, 324, 449
- prefix operator, 193
- preprocessor, 206
- preprocessor directive, 15, 25, 79, 87, 89, 93, 96, 327, 328, 491
- prev\_permutation(), 378
- prevent casts, 25
- preventing member function usage, 164
- previous element, 322
- primitive value, 199
- printf(), 22, 34, 77
- priority queue data structure, 232
- priority rules, 232, 498, 500
- priority\_queue, 232, 234
- priority\_queue::empty(), 234
- priority\_queue::pop(), 234

- priority\_queue::push(), 234
- priority\_queue::size(), 234
- priority\_queue::top(), 234
- private, 43, 119, 208, 212, 263, 268, 431, 440, 456, 484, 493, 499
- private assignment operator, 286
- private constructors, 164
- private copy constructor, 286
- private derivation, 262
- private members, 306, 435
- private static data member, 209
- problem analysis, 260
- procbuf, 15
- procedural approach, 20
- process ID, 467, 469
- processing files, 107
- profiler, 130, 225
- program development, 260
- Prolog, 19
- promoting a type, 162
- promotions, 434
- property, 220
- protected, 43, 101, 103, 268, 456, 492
- protected derivation, 262
- protocol, 276
- prototypes, 334
- prototyping, 18
- ptrdiff\_t, 449, 484
- public, 43, 120, 208, 211, 268, 440, 486, 499
- public derivation, 262
- public static data members, 208
- pubseekoff(), 289
- pure virtual functions, 276, 431
- pure virtual member functions, 292
- push\_back(), 325
- push\_front(), 325
- push\_heap(), 411
- qsort(), 448
- qualification conversion, 422, 424
- qualification conversions, 434
- queue, 218, 231, 422
- queue data structure, 231
- queue::back(), 231
- queue::empty(), 231
- queue::front(), 231
- queue::pop(), 232
- queue::push(), 232
- queue::size(), 232
- radix, 83, 84, 290
- random, 224, 235
- random access, 324
- random number generator, 380
- random\_access\_iterator, 449
- random\_shuffle(), 379
- RandomAccessIterator, 450, 484
- RandomAccessIterators, 324, 449
- range of values, 222
- rbegin(), 322
- read and write to a stream, 111
- read beyond end-of-file, 81
- read first, test later, 108
- read from a container, 324
- read from memory, 97
- reading a string, 66
- reading and writing fstreams, 111
- real numbers, 498
- real part, 258, 259
- realloc(), 151
- recipe, 415
- recompilation, 263
- redefining member functions, 265
- redirection, 109, 460, 466, 479
- Redirector, 470
- Redirector::bad\_stream, 471
- reduce typing, 237, 247
- reference, 205, 271, 273
- reference data members, 163
- reference operator, 123
- reference parameter, 133
- reference to the current object, 158
- reference: to template class, 433
- references, 36
- regular expression, 490, 492
- regular expressions, 497
- reinterpret\_cast<type>(expression), 24
- relational function object, 319, 321
- relational operations, 319, 482
- relative address, 298
- relative position, 106
- remove(), 381
- remove\_copy(), 382
- remove\_copy\_if(), 384
- remove\_if(), 383, 482
- rend(), 322
- renew operator, 142, 143
- replace(), 384
- replace\_copy(), 385
- replace\_copy\_if(), 387
- replace\_if(), 386
- repositioning, 88, 95
- resizing strings, 66
- responsibility of the programmer, 111, 222, 226, 231, 234, 235, 252, 333, 353, 485
- restricted functionality, 272
- retrieval, 237
- retrieve the type of objects, 283
- return, 23, 165, 194

- return type, 416
- return value, 23, 205, 415
- reusable software, 101
- Reverse Polish Notation, 251
- reverse(), 388
- reverse\_copy(), 388
- reverse\_iterator, 223, 228, 236, 242, 248, 431
- reversed sorting, 319
- reversed\_iterator, 322
- right shift, 238
- right-hand, 219, 221
- right-hand value, 184
- rotate(), 389
- rotate\_copy(), 390
- RPN, 251
- rule of thumb, 27, 35, 130, 139, 143, 199, 213, 224, 263, 298, 429, 430
- rule section, 498
- rules section, 491
- run-time, 273, 283, 294
- run-time error, 180
- run-time type identification, 283
- rvalue, 184, 193, 239, 324, 332, 333, 486
- scalar numerical types, 253
- scalar type, 258
- scan(), 465
- scanf(), 94
- scanner, 488
- scanner generator, 488
- scanner: as parser member, 503
- scientific notation, 84
- scope resolution operator, 32, 47, 199, 212, 266, 269, 279, 297, 305
- scope rules, 416
- search(), 391
- search\_n(), 392
- second, 220
- seek before begin of file, 89, 96
- seek beyond end of file, 89, 96
- seek\_dir, 310
- seekg(), 97, 112
- segmentation fault, 332
- select(), 477
- self-destruction, 156
- semantic value, 499
- semantical correctness, 430
- semantical value union, 500
- semicolon, 420
- sequential containers, 218
- set, 246
- set constructors, 247
- set: member functions, 248
- set::begin(), 248
- set::clear(), 248
- set::count(), 248, 249
- set::empty(), 248
- set::end(), 248
- set::equal\_range(), 248
- set::erase(), 248
- set::find(), 248
- set::insert(), 248
- set::lower\_bound(), 248
- set::rbegin(), 248
- set::rend(), 249
- set::size(), 249
- set::swap(), 249
- set::upper\_bound(), 249
- set\_difference(), 393
- set\_intersection(), 394
- set\_new\_handler(), 150
- set\_symmetric\_difference(), 396
- set\_union(), 397
- setg(), 456
- setjmp(), 165, 168, 172
- setjmp(): alternative to, 170
- setjmp(): avoid, 170
- setstate(): with streams, 83
- setup.exe, 18
- shrink arrays, 142, 143
- shuffling, 335
- side effect, 76
- side-effects, 206
- sigh of relief, 15
- SIGKILL, 478
- signal, 468
- significant digits, 86
- SIGTERM, 478
- silently ignored, 238, 247
- sin(), 21, 259
- sinh(), 259
- size specification, 210
- size\_t, 196, 198, 200
- sizeof, 17, 140
- skeleton algorithm, 469
- skeleton program, 440
- snext(), 288
- socket, 451
- sockets, 79
- software design, 101
- sort by multiple hierarchical criteria, 402
- sort(), 319, 324, 398
- sort\_heap(), 412
- sorted collection of value, 249
- sorted collection of values, 246
- sorting, 335
- Spartan, 486
- special containers, 219
- specialization parameter list, 446



- specialized constructor, 195
- specification file: with long lines, 499
- split buffer, 105
- sprintf(), 87
- sputc(), 289
- sqrt(), 259
- sscanf(), 93
- stable\_partition(), 399
- stable\_sort(), 302, 400
- stack, 160, 218, 251, 489, 494
- stack constructors, 252
- stack data structure, 251
- stack operations, 204
- stack::empty(), 252
- stack::pop(), 252
- stack::push(), 252
- stack::size(), 252
- stack::top(), 252
- stand alone functions, 180
- standard namespace, 22
- standard output, 489
- Standard Template Library, 15, 218, 313
- standard type conversions, 424
- stat(), 42
- state flags, 182
- state of I/O streams, 77, 80
- static, 20, 47, 208
- static binding, 273, 274
- static data member, 306
- static data members, 208, 448
- static data members: initialization, 209
- static inline member functions, 212
- static local variables, 251
- static member function, 197, 275
- static member functions, 211
- static members, 208, 302, 438
- static members: definitions, 438
- static object, 120
- static private members, 436
- static type checking, 283
- static type identification, 283
- static\_cast, 191
- static\_cast<type>(expression), 24
- std, 22
- std::bad\_cast, 285, 312
- stderr, 33
- stdin, 33
- STDIN\_FILENO, 474
- stdio.h, 26
- stdlib.h, 448
- stdout, 33
- STDOUT\_FILENO, 454
- step-child, 468
- step-parent, 468
- STL, 15, 218, 313
- storage, 237
- storing data, 224
- str...(), 140
- strcasecmp(), 314
- strdup(), 140, 151
- stream, 106, 287
- stream mode, 291
- streambuf, 77, 80, 81, 90, 101, 103, 106, 108, 116, 287, 327, 451, 455, 459, 461, 465
- streambuf::eback(), 103, 455, 463
- streambuf::egptr(), 103, 455, 463
- streambuf::epptr(), 453
- streambuf::gbump(), 104
- streambuf::gptra(), 104, 455, 463
- streambuf::gpump(), 458
- streambuf::in\_avail(), 101
- streambuf::overflow(), 102, 105, 288, 451, 454
- streambuf::pbackfail(), 104, 288
- streambuf::pbase(), 105, 453
- streambuf::pbump(), 105, 454
- streambuf::pptr(), 105, 453
- streambuf::pubseekoff(), 103
- streambuf::pubseekpos(), 103
- streambuf::pubsetbuf(), 103
- streambuf::pubsync(), 102
- streambuf::sbumpc(), 102, 458
- streambuf::seekoff(), 106, 289, 459
- streambuf::seekpos(), 106, 289, 459
- streambuf::setbuf(), 106, 289
- streambuf::setg(), 104, 455
- streambuf::setp(), 106, 453
- streambuf::sgetc(), 102
- streambuf::sgetn(), 102, 458
- streambuf::showmanyc(), 104, 288
- streambuf::snextc(), 102
- streambuf::sputback(), 102
- streambuf::sputc(), 102
- streambuf::sputn(), 102
- streambuf::streambuf(), 103
- streambuf::sungetc(), 102
- streambuf::sync(), 106, 289, 452, 453
- streambuf::uflow(), 102, 105, 288
- streambuf::underflow(), 105, 288, 462
- streambuf::xsgetn(), 102, 105, 288, 456
- streambuf::xspn(), 102, 106, 289
- streams: associating, 116
- streamsize, 101
- string, 57, 188, 288
- string appends, 60
- string assignment, 58
- string comparisons, 59
- string constructors, 67
- string elements, 59

- string erasing, 63
- string extraction, 94
- string initialization, 58
- string insertions, 61
- string operators, 67
- string pointer dereferencing operator, 59
- string range checking, 59
- string replacements, 62
- string searches, 64
- string size, 65
- string swapping, 63
- string to ASCII-Z conversion, 58
- string(char const \*), 320
- string: as union member, 500
- string::append(), 69
- string::assign(), 69
- string::at(), 59, 69
- string::begin(), 67, 219
- string::c\_str(), 70
- string::capacity(), 69
- string::compare(), 59, 69
- string::copy(), 70
- string::data(), 70
- string::empty(), 65, 70
- string::end(), 67, 219
- string::erase(), 70, 71
- string::find(), 71
- string::find\_first\_not\_of(), 72
- string::find\_first\_of(), 71
- string::find\_last\_not\_of(), 72
- string::find\_last\_of(), 72
- string::getline(), 73
- string::insert(), 73
- string::iterator, 303
- string::length(), 73
- string::max\_size(), 73
- string::npos, 57, 66, 69
- string::rbegin(), 67
- string::rend(), 67
- string::replace(), 74
- string::resize(), 74
- string::rfind(), 74
- string::size(), 75
- string::size\_type, 57, 68
- string::substr(), 75
- string::swap(), 75
- stringstream, 15
- strlen(), 65
- strongly typed, 415
- Stroustrup, 35
- strstream, 15
- struct, 30, 120, 151
- substrate, 156
- Substrings, 64
- substrings, 64
- subtraction, 315
- super specialization, 448
- superset, 428
- swap area, 150
- swap(), 402
- swap\_ranges(), 403
- swapping, 335
- Swiss army knife, 267
- switching files, 489
- symbol area, 491
- symbolic constants, 34
- symbolic name, 454
- syntactical elements, 166
- system call, 15, 451, 467, 473, 477, 478
- system(), 466, 467
- tags, 449
- TCP/IP stack, 101
- tellg(), 112
- template, 76, 218, 313, 414, 425, 431, 445
- template <...>, 415
- template argument deduction, 420, 423, 428
- template argument deduction: not required, 424
- template class, 422, 430, 438, 445, 482
- template class constructor, 434
- template class parameters, 449
- template class specialization, 445
- template class specialization interface, 446
- template class specializations, 444
- template class type parameters, 434
- template class: construction, 431
- template class: conversion to a base class, 423
- template class: default arguments, 432
- template class: fully specialized, 448
- template class: instantiation, 433
- template class: pointer to, 433
- template class: reference to, 433
- template class: specialization, 445
- template classes, 448
- template classes: having multiple parameters, 446
- template explicit arguments, 427
- template explicit specialization, 425
- template function, 415
- template function declarations, 425
- template function: address of, 417
- template function: not instantiated, 418
- template function: preferred, 429
- template functions: overloading, 426
- template functions: specialization, 427
- template instantiation declaration, 426
- template member function, 435
- template member functions, 448

- Template Method, 469
- template non-type parameter, 415
- template objects, 448
- template parameter, 446
- template parameter list, 415, 437, 443
- template parameters, 431, 442
- template parameters: identical types, 423
- template type parameter, 415
- template type parameters, 431
- template: declaration, 420
- template: parameter conversions, 421
- template<>, 445, 446
- templates in classes, 442
- templates: ambiguities, 427
- templates: and friends, 435
- terminal symbols, 500
- testing the 'open' status, 90, 97
- text files, 88, 112
- this, 156–158, 197, 211, 212, 275, 302, 318
- throw, 166, 172
- throw([type1 [, type2, type3, ...]]), 180
- throw: copy of objects, 172
- throw: empty, 175, 177
- throw: function return values, 175
- throw: local objects, 174
- throw: pointer to a dynamically generated object, 174
- throw: pointer to a local object, 174
- tie(), 108
- token, 251
- token indicators, 500
- token section, 498
- tokens, 497
- top, 252
- top-down, 260
- toString(), 176
- transform(), 318, 320, 404
- traverse containers, 324
- traverse containers, 324
- true, 41, 68, 82, 90, 97, 219, 321, 361, 371, 491
- truth value, 321
- try, 166, 171, 177, 178
- try block: destructors in, 179
- try block: ignoring statements, 179
- two types, 237
- Type, 220
- type cast, 201, 272, 353, 418, 424
- type checking, 22
- Type complex::imag(), 259
- Type complex::real(), 259
- type conversions, 428
- type name, 285
- type of the pointer, 272
- type parameter, 416
- type safe, 87, 93, 141
- type safety, 77
- type specification, 258
- type-safe, 34
- type\_info, 286
- typedef, 30, 76, 221, 237, 247, 402
- typedefs: nested, 441
- typeid, 283, 285
- typeid: argument, 287
- typename, 415
- typename: required, 417
- types of iterators, 324
- types.h, 484
- types: depending on template parameters, 430
- types: without values, 312
- typing effort, 221
- unary function, 321
- unary function objects, 321
- unary minus, 498
- unary not, 482
- unary operator, 318, 320, 482
- unary predicate, 342
- unbound friends, 437
- undefined reference, 276, 429
- Unicode, 42
- unimplemented member functions, 164
- union, 30, 419
- union fields, 500
- union: and constructors, 500
- union: without objects, 500
- unique(), 406
- unique\_copy(), 407
- Unix, 109, 150, 467, 481, 497, 504
- unresolved references, 504
- upper\_bound(), 408
- use of inline functions, 129
- using, 138
- using inline functions, 130
- using namespace std, 22
- using namespace std;, 15
- using-declaration, 48
- using-directive, 49
- value, 237
- value parameter, 174, 421
- value return type, 185
- value type, 253
- value-retrieval, 185
- value\_type, 237, 247
- variable number of arguments, 198
- vector, 218, 221, 322, 431
- vector constructors, 221
- vector: member functions, 222

- vector::back(), 222
- vector::begin(), 222
- vector::clear(), 222
- vector::empty(), 222
- vector::end(), 222
- vector::erase(), 222
- vector::front(), 223
- vector::insert(), 223
- vector::pop\_back(), 223
- vector::push\_back(), 223
- vector::rbegin(), 223
- vector::rend(), 223
- vector::resize(), 223
- vector::size(), 223
- vector::swap(), 223
- vform(), 15, 88, 465
- viable functions, 428
- virtual, 273, 276, 451, 490
- virtual base class, 279
- virtual derivation, 279
- virtual destructor, 275, 276, 293
- virtual member function, 273, 283
- virtual member functions, 287
- virtual table, 499
- visibility: nested classes, 303
- visible, 428
- visit all elements in a map, 243
- void, 25
- void \*, 178, 196, 198, 200, 419
- volatile, 422
- vpointer, 294
- vscan(), 465
- vscanf(), 94
- vsnprintf(), 465
- vtable, 294
  
- waitpid(), 478
- wchar\_t, 42, 76
- white space, 85
- wild pointer, 152, 174, 329
- WINDOWS, 112
- wrapper, 149, 292, 489
- wrapper class, 79, 194, 233, 270
- write beyond end of file, 89
- write to a container, 324
- write to memory, 91
  
- yacc, 488
- Yodl, 14
- YY\_, 498
- yy\_buffer\_state, 494
- YY\_DECL, 490
- yy\_delete\_buffer(), 495
- yy\_switch\_to\_buffer(), 494
- yyFlexLexer, 489, 490, 492, 498
- yyFlexLexer::set\_debug(), 497
- yyFlexLexer::yylex(), 490
- yyin, 491
- yylen, 492
- YYLeng(), 493
- yylex(), 490
- yylineno, 493, 494
- yyval, 499
- yyout, 491
- yytext, 492
- YYText(), 492, 500
- yywrap(), 504
  
- zombi, 468, 478