C++ Annotations Version 6.5.0

Frank B. Brokken
Computing Center, University of Groningen
Nettelbosje 1,
P.O. Box 11044,
9700 CA Groningen
The Netherlands
Published at the University of Groningen
ISBN 90 367 0470 7

1994 - November 2006

Abstract

This document is intended for knowledgeable users of **C** (or any other language using a **C**-like grammar, like **Perl** or **Java**) who would like to know more about, or make the transition to, **C++**. This document is the main textbook for Frank's **C++** programming courses, which are yearly organized at the University of Groningen. The **C++** Annotations do not cover all aspects of **C++**, though. In particular, **C++**'s basic grammar, which is, for all practical purposes, equal to **C**'s grammar, is not covered. For this part of the **C++** language, the reader should consult other texts, like a book covering the **C** programming language.

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

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

in files having names starting with `cplusplus` (A4 paper size). Files having names starting with 'cplusplusus' are intended for the US *legal* paper size.

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

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

Contents

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: Template functions: using *molds* for type independent functions.
- Chapter 19: Template classes: using *molds* for type independent classes.
- Chapter 20: Several examples of programs written in **C++**.

Chapter 2

Introduction

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

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

The version number of the **C++** Annotations (currently 6.5.0) 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 under the `GNU General Public License`[2].

The **C++ Annotations** were typeset using the `yodl`[3] formatting system.

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

Frank B. Brokken
Computing Center, University of Groningen
Nettelbosje 1,
P.O. Box 11044,
9700 CA Groningen
The Netherlands
(email: f.b.brokken@rug.nl)

In this chapter a first impression of **C++** is presented. A few extensions to **C** are reviewed and the

[1]http://publications.gbdirect.co.uk/c_book/
[2]http://www.gnu.org/licenses/
[3]http://yodl.sourceforge.net

concepts of object based and object oriented programming (OOP) are briefly introduced.

2.1 What's new in the C++ Annotations

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

- Version 6.5.0 changed `unsigned` into `size_t` where appropriate, and explicitly mentioned `int`-derived types like `int16_t`. In-class member function definitions were moved out of (below) their class definitions as `inline` defined members. A paragraphs about implementing pure virtual member functions was added. Various bugs and compilation errors were fixed.
- Version 6.4.0 added a new section (19.11.2) further discussing the use of the `template` keyword to distinguish types nested under template classes from template members. Furthermore, *Sergio Bacchi* `s dot bacchi at gmail dot com` did an impressive job when translating the Annotations into Portuguese. His translation (which may lag a distribution or two behind the latest verstion of the Annotations) may also be retrieved from `ftp://ftp.rug.nl/contrib/frank/documents/annotations`.
- Version 6.3.0 added new sections about anonymous objects (section 6.2.1) and type resolution with template classes (section 19.11.1). Also the description of the template parameter deduction algorithm was rewritten (cf. section 18.2.4) and numerous modifications required because of the compiler's closer adherence to the **C++** standard were realized, among which exception rethrowing from constructor and destructor function try blocks. Also, all textual corrections received from readers since version 6.2.4 were processed.
- In version 6.2.4 many textual improvements were realized. I received extensive lists of typos and suggestions for clarifications of the text, in particular from Nathan Johnson and from Jakob van Bethlehem. Equally valuable were suggestions I received from various other readers of the **C++** annotations: all were processed in this release. The **C++** content matter of this release was not substantially modified, compared to version 6.2.2.
- Version 6.2.2 offers improved implementations of the configurable template classes (sections 20.7.3 and 20.7.4).
- Version 6.2.0 was released as an Annual Update, by the end of May, 2005. Apart from the usual typo corrections several new sections were added and some were removed: in the Exception chapter (8) a section was added covering the standard exceptions and their meanings; in the chapter covering static members (10) a section was added discussing `static const` data members; and the final chapter (20) covers configurable template classes using *local context structs* (replacing the previous `ForEach, UnaryPredicate` and `BinaryPredicate` classes). Furthermore, the final section (covering a **C++** parser generator) now uses **bisonc++**, rather than the old (and somewhat outdated) **bison++** program.
- Version 6.1.0 was released shortly after releasing 6.0.0. Following suggestions received from Leo Razoumov`<LEOR@winmain.rutgers.edu>` and Paulo Tribolet, and after receiving many, many useful suggestions and extensive help from Leo, navigatable `.pdf` files are from now on distributed with the **C++** Annotations. Also, some sections were slightly adapted.
- Version 6.0.0 was released after a full update of the text, removing many inconsistencies and typos. Since the update effected the Annotation's full text an upgrade to a new major version seemed appropriate. Several new sections were added: overloading binary operators (section 9.6); throwing exceptions in constructors and destructors (section 8.8); function `try`-blocks (section 8.9); calling conventions of static and global functions (section 10.2.1) and virtual constructors (section 14.10). The chapter on templates was completely rewritten and split into

two separate chapters: chapter 18 discusses the syntax and use of template *functions*; chapter 19 discusses template *classes*. Various concrete examples were modified; new examples were included as well (chapter 20).

- In version 5.2.4 the description of the *random_shuffle* generic algorithm (section 17.4.39) was modified.

- In version 5.2.3 section 2.5.10 on local variables was extended and section 2.5.11 on function overloading was modified by explicitly discussing the effects of the **const** modifier with overloaded functions. Also, the description of the `compare()` function in chapter 4 contained an error, which was repaired.

- In version 5.2.2 a leftover in section 9.4 from a former version was removed and the corresponding text was updated. Also, some minor typos were corrected.

- In version 5.2.1 various typos were repaired, and some paragraphs were further clarified. Furthermore, a section was added to the *template* chapter (chapter 18), about creating several `iterator` types. This topic was further elaborated in chapter 20, where the section about the construction of a reverse iterator (section 20.5) was completely rewritten. In the same chapter, a *universal text to anything convertor* is discussed (section 20.6). Also, `LaTeX`, `PostScript` and `PDF` versions fitting the *US-letter* paper size are now available as `cplusplus`**us** versions: `cplusplusus.latex, cplusplusus.ps` and `cplusplus.pdf`. The *A4-paper* size is of course kept, and remains to be available in the `cplusplus.latex, cplusplus.ps` and `cpluspl.pdf` files.

- Version 5.2.0 was released after adding a section about the `mutable` keyword (section 6.5), and after thoroughly changing the discussion of the `Fork()` abstract base class (section 20.3). All examples should now be up-to-date with respect to the use of the `std` namespace.

- However, in the meantime the Gnu `g++` compiler version 3.2 was released[4]. In this version extensions to the abstract containers (see chapter 12) like the hash_map (see section 12.3.11) were placed in a separate namespace, `__gnu_cxx`. This namespace should be used when using these containers. However, this may break compilations of sources with `g++`, version 3.0. In that case, a compilation can be performed conditionally to the 3.2 and the 3.0 compiler version, defining `__gnu_cxx` for the 3.2 version. Alternatively, the *dirty trick*

```
#define __gnu_cxx std
```

  can be placed just before header files in which the `__gnu_cxx` namespace is used. This might eventually result in name-collisions, and it's a dirty trick by any standards, so please don't tell anybody I wrote this down.

- Version 5.1.1 was released after modifying the sections related to the `fork()` system call in chapter 20. 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

[4] `http://www.gnu.org`

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 and reordered, so that 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 default pagesize might be a better choice. In that case, remove the `a4paper,twoside` 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`[5] 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 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

[5] ftp://research.att.com/dist/c++std/WP/

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 is 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.*.* (replace the stars by any applicable number) were not logged.

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

2.2 C++'s history

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 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 its extension. This holds true for Borland's and Microsoft's **C++** compilers, which assume a **C++** source for an extension `.cpp`. The Gnu compiler `g++`, which is available on many Unix platforms, assumes for **C++** the extension `.cc`.

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

2.2.1 History of the C++ Annotations

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

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

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

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`[6].

[6] http://yodl.sourceforge.net

> The **C++** Annotations are freely distributable. Be sure to read the `legal notes`[7].
>
> **Reading the annotations beyond this point implies that you are aware of these notes and that you agree with them.**

If you like this document, tell your friends about it. Even better, let us know by sending email to `Frank`[8].

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.csci.csusb.edu/dick/c++std/cd2/index.html`: offers a version of the 1996 working paper of the **C++** ANSI/ISO standard.

2.2.2 Compiling a C program using a C++ compiler

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

- In **C**, `sizeof('c')` equals `sizeof(int)`, `'c'` being any ASCII character. The underlying philosophy is probably that `chars`, 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), the two function calls

  ```
  somefunc(10);
  ```

  and

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

  may be handled by quite separate functions: **C++** distinguishes functions not only by their names, but also by their argument types, which are different in these two calls: one call using an `int` argument, the other one using a `char`.

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

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

  in **C** means that a function `func()` exists, which returns no value. 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: passing arguments to it results in a compile-time error.

[7] legal.shtml
[8] mailto:f.b.brokken@rug.nl

2.2.3 Compiling a C++ program

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`).

2.2.3.1 C++ under MS-Windows

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

When visiting the above URL to obtain a free `g++` compiler, click on `install now`. This will download the file `setup.exe`, which can be run to install `cygwin`. The software to be installed can be downloaded by `setup.exe` from the internet. There are alternatives (e.g., using a CD-ROM), which are described on the Cygwin page. Installation proceeds interactively. The offered defaults are 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 an already available compiler. The compiler's webpage (mentioned above) contains detailed instructions on how to proceed. In our experience building a new compiler within the Cygnus environment works flawlessly.

2.2.3.2 Compiling a C++ source text

In general, the following command is used to compile a **C++** source file '`source.cc`':

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

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

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

If a mere compilation is required, the compiled module can be 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`[9] program a maintenance script can be used to assist in the construction and maintenance of **C++** programs. A generic `icmake` maintenance script (`icmbuild`) is available as well. Alternatively, the standard `make` program can be used to maintain **C++** programs. It is strongly advised to start using maintenance scripts or programs early in the study of the **C++** programming language. Alternative approaches were implemented by former students, e.g., `lake`[10] by Wybo Wiersma and `ccbuild`[11] by Bram Neijt.

[9] ftp://ftp.rug.nl/contrib/frank/software/linux/icmake
[10] http://nl.logilogi.org/MetaLogi/LaKe
[11] http://ccbuild.sourceforge.net/

2.3 C++: advantages and claims

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

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

Which of these allegations are true? Originally, our impression was that the **C++** language was a little overrated; the same holding true for the entire object-oriented programming (OOP) approach. The enthusiasm for the **C++** language resembles the once uttered allegations about Artificial-Intelligence (AI) languages like Lisp and Prolog: these languages were supposed to solve the most difficult AI-problems 'almost without effort'. Obviously, too promising stories about any programming language must be overdone; in the end, each problem can be coded in any programming language (say 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 implement an efficient and understandable solution for a programming problem'.

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

- The development of new programs while existing code is reused can also be realized in **C** by, e.g., using function libraries. Functions can be collected in a library and need not be re-invented with each new program. **C++**, however, offers specific syntax possibilities for code reuse, apart from function libraries (see 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. In **C++** these facilities are augmented by defining data types which are completely 'self supporting', taking care of, e.g., their memory management automatically (without having to resort to an independently operating memory management system as used in, e.g., **Java**).
- 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 (allocating the memory or aborting the program when the memory pool is exhausted). However, with `malloc()` like functions it is easy to err: miscalculating the required number of bytes in a `malloc()` call is a frequently occurring error. Instead, **C++** offers facilities for allocating memory in a somewhat safer way, through its `operator new`.
- Concerning 'bug proneness' we can say that **C++** indeed uses stricter type checking than **C**. However, most modern **C** compilers implement 'warning levels'; it is then the programmer's choice to disregard or 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, making it far easier to realize 'data hiding' in **C++** than in **C**.

C++ in particular (and OOP in general) is of course not *the* solution to all programming problems. However, the language *does* offer various new and elegant facilities which are worthwhile investigating. At the same time, the level of grammatical complexity of **C++** has increased significantly compared to **C**. This may be considered a serious disadvantage of the language. Although we got used to this increased level of complexity over time, the transition wasn't fast or painless. With the **C++** 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++**. It is our hope that you like this document and may benefit from it. Enjoy and good luck on your journey into **C++**!

2.4 What is Object-Oriented Programming?

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

In contrast (or maybe better: in addition) to this, an object-based approach identifies **keywords** in a 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 an entity: data types and functions to manipulate the data. As an example of an object oriented approach, an illustration follows:

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

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

- The owner and the mechanics can be represented 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 some *extra* functionality: the number of transactions (sales) and the bonus per transaction.

  In the hierarchy of objects we would define the dependency between the first two objects by letting the car salesmen be 'derived' from the owner and mechanics.

- Finally, there are the second-hand car purchasers. These share the functionality of the salesmen except for 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 are further illustrated in Figure 2.1.

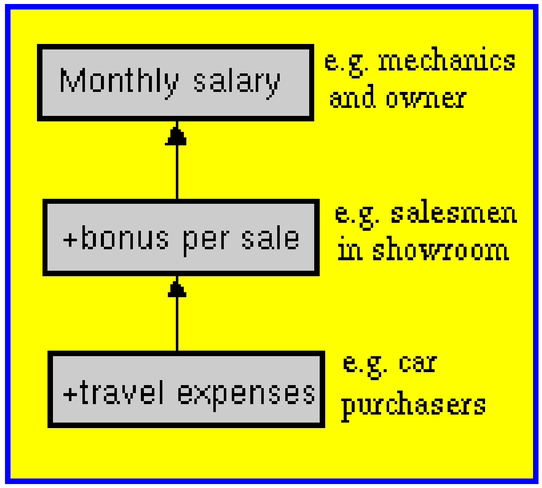

Figure 2.1: Hierarchy of objects in the salary administration.

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 are usually available to objects of other classes. As we will see, classes tend to *hide* their properties in such a way that they are not directly modifiable by the outside world. Instead, dedicated functions are used to reach or modify the properties of objects. Also, these objects tend to be *self-contained*. They *encapsulate* all the functionality and data required to perform their tasks and to uphold the object's integrity.

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*, but does not want to lose the capability of using the standard `sin()` function, operating on *radians*.

Namespaces are covered extensively in section 3.7. For now it should be noted that most compilers require the explicit declaration of a *standard namespace*: `std`. So, unless otherwise indicated, it is stressed that all examples in the Annotations now implicitly use the

```
using namespace std;
```

declaration. So, if you actually intend to compile 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, covering
        multiple lines
    */
}
```

Despite the example, it is advised *not* to use **C** type comment inside the body of **C++** functions. At times you will temporarily want to suppress sections of existing code. In those cases it's very practical to be able to use standard **C** comment. If such suppressed code itself contains such comment, it would result in nested comment-lines, resulting in compiler errors. Therefore, the rule of thumb is not to use **C** type comment inside the body of **C++** functions.

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`, and using `0` is actually preferred when indicating null-pointer values.

2.5.4 Strict type checking

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

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

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. The error is of course the missing `#include <stdio.h>` directive.

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
}
```

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. Apart from the **C** style cast (now deprecated) **C++** also supports the *function call* notation:

```
typename(expression)
```

This function call notation is not actually a cast, but the request to the compiler to construct an (anonymous) variable of type `typename` from the expression `expression`. This form is actually very often used in **C++**, but should *not* be used for casting. 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 further down 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.

2.5.5.1 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 `d` is of type `double` and `a` and `b` are `int`-type variables. In that situation, computing the floating point quotient of `a` and `b` requires a cast:

```
d = static_cast<double>(a) / b;
```

If the cast is omitted, the division operator will cut-off the remainder, as its operands are `int` expressions. Note that the division should be placed outside of the cast. If not, the (integer) division will be performed before the cast has a chance to convert the type of the operand to `double`. 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;
```

This statement actually evaluates to:

```
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);
```

2.5.5.2 The ‘const_cast’-operator

The `const_cast<type>(expression)` operator is used to undo 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));
```

2.5.5.3 The ‘reinterpret_cast’-operator

The `reinterpret_cast<type>(expression)` operator is used to reinterpret pointers. For example, using a `reinterpret_cast<>()` the individual bytes making up a `double` value can easily be reached. 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 will not eventually be the underlying cause for a program to misbehave.

2.5.5.4 The 'dynamic_cast'-operator

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

2.5.6 The 'void' parameter list

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

```
void func();
```

means that the argument list of the declared function is not prototyped: the compiler will not warn against improper argument usage. In **C**, to declare a function having no arguments, the keyword `void` is used:

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

As **C++** enforces strict type checking, an empty parameter list indicates the absence of any parameter. The keyword `void` can thus be omitted: in **C++** the above two function 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 Using standard C functions

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

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

```
extern "C" void *xmalloc(size_t 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>
}
```

Although these two approaches can be used, they are actually seldomly encountered in **C++** sources. We will encounter a more frequently used method to declare external **C** functions in the next section.

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.,
    void *xmalloc(size_t 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++**.

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

```
#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_ has been defined and consequently all information between the #ifndef and #endif directives is skipped by the compiler.

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

Apart from all this, the custom has evolved to give **C** header files the extension .h, and to give C++ header files *no* extension. For example, the standard *iostreams* cin, cout and cerr are available after including the 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 6.6 the preferred organization of **C++** header files is discussed.

2.5.10 Defining local variables

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

Furthermore, local variables can be defined inside 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 older 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 then 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 inside 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 and often more efficient code than defining variables at the beginning of compound statements. We suggest the following rules of thumb for defining local variables:

- Local variables should be created at 'intuitively right' places, such as in the example above. This does not only entail the for-statement, but also all situations where a variable is only needed, say, half-way through the function.

- More in general, variables should be defined in such a way that their scope is as *limited* and *localized* as possible. Local variables are not necessarily defined anymore at the beginning of functions, following the first `{`.
- It is considered good practice to *avoid global variables*. It is fairly easy to lose track of which global variable is used for what purpose. In **C++** global variables are seldomly required, and by localizing variables the well known phenomenon of using the same variable for multiple purposes, thereby invalidating each individual purpose of the variable, can easily be avoided.

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

```
#include <stdio.h>

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

        case EOF:
            printf("Saw EOF\n");
        break;

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

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

The same approach can be used with `if` and `while` statements: a variable that is defined in the condition part of an `if` and `while` statement is available in their nested statements. However, one should realize that:

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

The latter point of attention should come as no big surprise: in order to be able to evaluate the logical condition of an `if` or `while` statement, the value of the variable must be interpretable as either zero (false) or non-zero (true). Usually this is no problem, but in **C++** *objects* (like objects of the type `std::string` (cf. chapter 4)) are often returned by functions. Such objects may or may not be interpretable as numerical values. If not (as is the case with `std::string` objects), then such variables can *not* be defined in the condition or expression parts of condition- or repetition statements. The following example will, therefore, *not* compile:

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

The above deserves further clarification. Often a variable can profitably be given local scope, but an extra check is required immediately following its initialization. Both the initialization and the test cannot be combined in one expression, but two nested statements are required. The following example will therefore *not* compile either:

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

If such a situation occurs, either use two nested `if` statements, or localize the definition of `int c` using a nested compound statement. Actually, other approaches are possible as well, like using *exceptions* (cf. chapter 8) and specialized functions, but that's jumping a bit too far ahead. At this point in our discussion, we can suggest one of the following approaches to remedy the problem introduced by the last example:

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

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

2.5.11 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 (and/or in their `const` attribute). An example is given below:

```
#include <stdio.h>

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

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

```
}

void show(char *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 identical 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:

- Do not use function overloading for functions doing conceptually different tasks. In the example above, the functions `show()` are still somewhat related (they print information to the screen).

  However, it is also quite possible to define two functions `lookup()`, one of which would find a name in a list while the other would determine the video mode. In this case the 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 `vidmode()`.

- **C++** does not allow identically named functions to differ only in their return value, as it is always the programmer's choice to either use 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()`. 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. Here, **C++** will call the function expecting an integer argument, which might not be what one expects.

- In chapter 6 the notion of `const` member functions will be introduced (cf. section 6.2). Here it is merely mentioned that classes normally have so-called member functions associated with them (see, e.g., chapter 4 for an informal introduction of the concept). Apart from overloading member functions using different parameter lists, it is then also possible to overload member functions by their `const` attributes. In those cases, classes may have pairs of identically named member functions, having identical parameter lists. Then, these functions are overloaded by their `const` attribute: one of these function must have the `const` attribute, and the other must not.

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");

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 unless explicitly specified in the call. 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);

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

```
// 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 function declarations in header files is incorrect: when the function is used in other sources the compiler will read the header file and not the function definition. Consequently, in those cases the compiler has no way to determine the values of default function arguments. Current compilers may generate errors when detecting default arguments in function definitions.

2.5.13 The keyword 'typedef'

The keyword `typedef` is still allowed in **C++**, but is not required anymore when defining `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. Here we encounter the first concrete example of an object: as previously was described (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
    int 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. With respect to this `struct` it should be noted that:

- The function `draw()` mentioned in the `struct` definition is a mere *declaration*. The actual code of the function, or in other words the actions performed by the function, are located elsewhere. We will describe the actual definitions of functions inside `structs` later (see section 3.2).
- The size of the `struct point` is equal to the size of its two `ints`. A function declared inside the structure does not affect its size. The compiler implements this behavior by allowing the function `draw()` to be known only in the context of a `point`.

The `point` structure could be used as follows:

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

a.x = 0;                // define first dot
a.y = 10;               // and draw it
a.draw();

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

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 behind 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 **C++** is further explored. The possibility to declare functions in `structs` is illustrated in various examples. The concept of a `class` is introduced.

3.1 More extensions to C in C++

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

3.1.1 The scope resolution operator ::

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

```
#include <stdio.h>

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. In **C++** the scope operator is used extensively, but it is seldomly used to reach a global variable shadowed by an identically named local variable. Its main purpose will be described in chapter 6.

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

Analogous 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 written to streams or read from them using the operators $<<$, called the *insertion operator* and $>>$, called the *extraction operator*. This is illustrated in the next example:

```
#include <iostream>

using namespace std;

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`. With respect to streams, please note:

- The standard 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 in which the standard streams are used.
- The streams `cout`, `cin` and `cerr` are variables of so-called *class*-types. Such variables are commonly called *objects*. Classes are discussed in detail in chapter 6 and are used extensively in **C++**.
- The stream `cin` extracts data from a stream and copies the extracted information to variables (e.g., `ival` in the above example) using the extraction operator (two consecutive > characters:

>>). We will describe later how operators in **C++** can perform quite different actions than what they are defined to do by the language, as is the case here. Function overloading has already been mentioned. In **C++** *operators* can also have multiple definitions, which is called *operator overloading*.

- The operators which manipulate `cin`, `cout` and `cerr` (i.e., >> and <<) also manipulate variables of different types. In the above example `cout << ival` results in the printing of an integer value, whereas `cout << "Enter a number"` results in the printing of a string. The actions of the operators therefore depend on the types of supplied variables.
- The *extraction operator* (>>) performs a so called *type safe* assignment to a variable by 'extracting' its value from a text-stream. Normally, the extraction operator will skip all *white space* characters that precede the values to be extracted.
- Special symbolic constants are used for special situations. The termination of a line written by `cout` is usually realized by inserting the `endl` symbol, rather than the string `"\n"`.

The streams `cin`, `cout` and `cerr` are 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 like `printf()` are not part of the **C** grammar, but were originally written by people who considered such functions important and collected them in a run-time library.

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*: new functionality can easily be added to 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, streams 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 grasped the possibilities of stream objects.

3.1.3 The keyword 'const'

The keyword `const` is very often seen in **C++** programs. Although `const` is part of the **C** grammar, in **C** `const` is used much less frequently.

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

```
int main()
{
    int const ival = 3;     // a constant int
                            // initialized to 3

    ival = 4;               // assignment produces
                            // an error message
}
```

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

Variables 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 use 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 like `*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`):

> *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 indentifier:

> "Buf is a const pointer to const characters"

This rule of thumb is especially useful in cases where confusion may occur. In examples of **C++** code, 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 we try 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's `C++ FAQ`[1] gives the same rule (paragraph 18.5) , in a similar context:

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

Marshal Cline's advice might be improved, though: You should start to read pointer definitions (and declarations) at the variable name, reading as far as possible to the definition's end. Once a closing parenthesis is seen, reading continues backwards from the initial point of reading, from right-to-left,

[1] http://www.parashift.com/c++-faq-lite/const-correctness.html

until the matching open-parenthesis or the very beginning of the definition is found. For example, consider the following complex declaration:

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

Here, we see:

- the variable `ip`, being a
- (reading backwards) modifiable pointer to an
- (reading forward) array of
- (reading backward) constant pointers to an
- (reading forward) array of
- (reading backward) modifiable pointers to constant characters

3.1.4 References

In addition to the well known ways to define variables, plain variables or pointers, **C++** allows 'references' to be defined as synonyms for variables. A reference to a variable is like an *alias*; the variable and the reference can both be used in statements involving the variable:

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

In the above example a variable `int_value` is defined. Subsequently a reference `ref` is defined, which (due to its initialization) refers to the same memory location as `int_value`. In the definition of `ref`, the reference operator & indicates that `ref` is not itself an 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 that increases the value of its argument by five but 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 also 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
}
```

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.

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. Nevertheless, pointers and references *should* be distinguished: once initialized, references can never refer to another variable, whereas the values of pointer variables can be changed, which will result in the pointer variable pointing to another location in memory. For example:

```
extern int *ip;
extern int &ir;

ip = 0;     // reassigns ip, now a 0-pointer
ir = 0;     // ir unchanged, the int variable it refers to
            // is now 0.
```

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

- In those situations where a called function does not alter its arguments of primitive types, a copy of the variables 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 values of its arguments, a pointer parameter is preferred. These pointer parameters should preferably be the initial parameters of the function. This is called 'return by argument'.

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

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

```
void by_reference(string const &str)
{
    cout << str;
}

int main ()
{
    int x = 7;
    string str("hello");

    by_pointer(&x);          // a pointer is passed
    by_reference(str);       // str is not altered
    return 0;                // x might be changed
}
```

References play an important role in cases where the argument will not be changed by the function, but where it is undesirable to use the argument to initialize the parameter. Such a situation occurs when a large variable, e.g., a `struct`, is passed as argument, or is returned by the function. In these cases the copying operation tends to become a significant factor, as the entire structure must be copied. So, in those cases references are preferred. 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 should be used. Consider the following example:

```
struct Person                   // some large structure
{
    char    name[80],
    char    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 &person(int index)
{
    return person[index];   // a reference is returned,
}                           // not a copy of person[index]

int main()
```

```
    {
        Person boss;

        printperson (boss);     // no pointer is passed,
                                // so variable won't be
                                // altered by the function
        printperson(person(5));
                                // references, not copies
                                // are passed here
        return 0;
    }
```

- Furthermore, it should be noted that there is yet another reason to use references when passing objects as function arguments: when passing a reference to an object, the activation of the so called *copy constructor* is avoided. Copy constructors will be covered in chapter 7.

References may result in extremely 'ugly' code. A function may 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 normally not be used. Nonetheless, there are situations where it is useful to return a reference. We have actually already seen an example of this phenomenon at our previous discussion of the streams. 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

Earlier it was mentioned that functions can be part of `structs` (see section 2.5.14). Such functions are called *member functions* or *methods*. This section discusses how to define such functions.

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

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

    void print();
};
```

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 basic types with several new types: the types bool, wchar_t, long long and long double (Cf. ANSI/ISO draft (1995), par. 27.6.2.4.1 for examples of these very long types). The type long long is merely a double-long long datatype. The type long double is merely a double-long double datatype. 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.

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

- E or e: the *exponentiation* character in floating point literal values. For example: 1.23E+3. Here, E should be pronounced (and iterpreted) as: *times 10 to the power*. Therefore, 1.23E+3 represents the value 1230.
- F can be used as *postfix* to a non-integral numerical constant to indicate a value of type float, rather than double, which is the default. For example: 12.F (the dot transforms 12 into a floating point value); 1.23E+3F (see the previous example. 1.23E+3 is a double value, whereas 1.23E+3F is a float value).
- L can be used as *prefix* to indicate a character string whose elements are wchar_t-type characters. For example: L"hello world".
- L can be used as *postfix* to an integral value to indicate a value of type long, rather than int, which is the default. Note that there is no letter indicating a short type. For that a static_cast<short>() must be used.
- U can be used as *postfix* to an integral value to indicate an unsigned value, rather than an int. It may also be combined with the postfix L to produce an unsigned long int value.

3.3.1 The data type 'bool'

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 then 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 filename 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 I suggest the following: if a function should inform its caller about the success or failure of its task, let the function return a `bool` value. If the function should return success or various types of errors, let the function return *enum* values, documenting the situation 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 data type 'wchar_t'

The `wchar_t` type is an extension of the `char` basic type, to accomodate *wide* character values, such as the *Unicode* character set. The `g++` compiler (version 2.95 or beyond) reports `sizeof(wchar_t)` as 4, which easily accomodates all 65,536 different *Unicode* character values.

Note that a programming language like **Java** has a data type `char` that is comparable to **C++**'s `wchar_t` type. **Java**'s `char` type is 2 bytes wide, though. On the other hand, **Java**'s `byte` data type is comparable to **C++**'s `char` type: one byte. Very convenient....

3.3.3 The data type 'size_t'

The `size_t` type is not really a built-in primitive data type, but a data type that is promoted by **POSIX** as a typename to be used for non-negative integral values. It is not a specific **C++** type, but also available in, e.g., **C**. It should be used instead of `unsigned int`. Usually it is defined implictly when a system header file is included. The header file 'officially' defining `size_t` in the context of **C++** is `cstddef`.

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

The type `size_t` should be used in all situations where non-negative integral values are intended. Sometimes functions explictly require `unsigned int` to be used. E.g., on amd-architectures the X-windows function `XQueryPointer` explicitly requires a pointer to a `unsigned int` variable as one of its arguments. In this particular situation a pointer to a `size_t` variable can't be used. This situation is exceptional, though. Usually a `size_t` can (and should) be used where unsigned values are intended.

Other useful bit-represented types also exists. E.g., `uns32_t` is guaranteerd to hold 32-bits unsigned values. Analogously, `int32_t` holds 32-bits signed values. Corresponding types exist for 8, 16 and 64 bits values. These types are defined in the header file `stdint.h`.

3.4 Keywords in C++

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

```
   and        const     float         operator static_cast    using
and_eq   const_cast       for               or      struct  virtual
   asm     continue    friend            or_eq      switch     void
  auto      default      goto          private    template volatile
bitand       delete        if        protected        this  wchar_t
 bitor           do    inline           public       throw    while
  bool       double       int         register        true      xor
 break dynamic_cast      long reinterpret_cast         try   xor_eq
  case         else   mutable           return     typedef
 catch         enum namespace            short      typeid
  char     explicit       new           signed    typename
 class       extern       not           sizeof       union
 compl        false    not_eq           static    unsigned
```

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

3.5 Data hiding: public, private and class

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

C++ has three special keywords which are related to data hiding: `private`, `protected` and `public`. These keywords can be used 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 to its member functions). The keyword `protected` is discussed in chapter 13, and is beyond the scope of the current discussion.

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

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

The data fields `d_name` and `d_address` are only accessible to the member functions which are defined in the `struct`: these are the functions `setName()`, `setAddress()` etc.. This results from the fact that the fields `d_name` and `d_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.d_name, "Knarf");  // error, name is private
```

Data hiding is realized as follows: the actual data of a `struct Person` are mentioned in the structure definition. The data are accessed by the outside world using special functions, 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 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 `name()` and `address()` return a `char const *`: the caller may not modify the strings to which the return values point.

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

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

char const *Person::name()
{
    return d_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., checking the validity of the data. In the above example `setName()` copies only up to 79 characters from its argument to the data member `name`, thereby avoiding array buffer overflow.

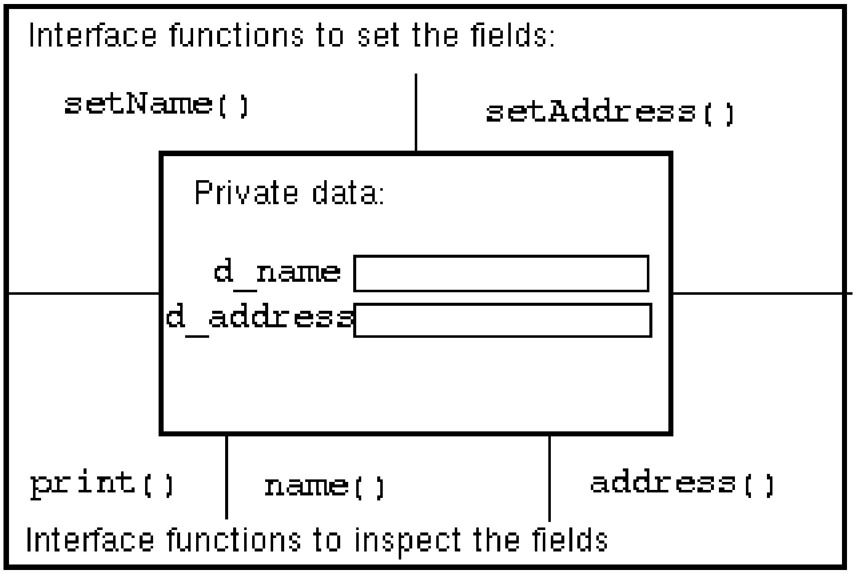

Figure 3.1: Private data and public interface functions of the class Person.

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 not require any modification of the program using `Person` structures. After recompilation and linking the new object module to a new library, the program will use the new `Person` structure.

Though data hiding can be realized with `structs`, more often (almost always) classes are used instead. A `class` refers to the same concept as a struct, except that a `class` uses private access by default, whereas structs use public access by default. 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, and the initial `private:` clause can be omitted. Our typographic suggestion for class names is to use a capital character as its first character, followed by the remainder of the name in lower case (e.g., `Person`).

3.6 Structs in C vs. structs in C++

Next 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];
        char address[80];
    } PERSON_;

    // some functions to manipulate PERSON_ structs

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

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

    // etc..
```

In **C++**, the declarations of the involved functions are 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();
            // etc..
        private:
            char d_name[80];
            char d_address[80];
    };
```

The `struct` argument is implicit in **C++**. A **C** function call such as:

```
    PERSON_ x;

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

becomes in **C++**:

```
    Person x;

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

3.7 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 usually solved by defining another name, e.g., the function name `cosDegrees()` is defined. **C++** offers an alternative solution: by allowing us to use *namespaces*. Namespaces can be considered as areas or regions in the code in which identifiers are defined which normally won't conflict with names already defined elsewhere.

Now that the ANSI/ISO standard has been implemented to a large degree in recent compilers, the use of namespaces is more strictly enforced than in previous versions of compilers. This 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 6.6.1 the construction of header files using entities from namespaces is discussed.

3.7.1 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 called '*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 be defined outside of their namespaces. This topic is discussed in section 3.7.4.1.

3.7.1.1 Declaring entities in namespaces

Instead of *defining* entities in a namespace, entities may also be *declared* in a namespace. This allows us to put all the declarations 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);
}
```

3.7.1.2 A closed namespace

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

Entities defined in the anonymous namespace are comparable to **C**'s `static` functions and variables. In **C++** the `static` keyword can still be used, but its use is more common 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 source files.

3.7.2 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;
}
```

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

However, in these cases an *abbreviated* form (just `cos()`) can be used by specifying a *using-declaration*. 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;
}
```

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.
}
```

3.7.2.1 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 where 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.

3.7.2.2 'Koenig lookup'

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

'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 the namespace in which the arguments are defined contains such a function, 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
*/
```

3.7.3 The standard 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.

3.7.4 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` is 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;
    }
```

- 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;
    }
```

- 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;
    }
```

- A `using` directive or directives can be used:

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

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

  Alternatively, two separate `using` directives could have been used:

```
    ...
    using namespace CppAnnotations;
    using namespace Virtual;
```

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

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, we may now use the construction

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

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

3.7.4.1 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 a member function `funny`, inside the namespace `CppAnnotations::Virtual`, returning a pointer to `CppAnnotations::Virtual::INT8`. After first defining everything inside the `CppAnnotations::Virtual` namespace, such a function could be defined as follows:

```
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] = (idx + 1) * (idx + 1);

            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` namespace 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)
        (*ip)[idx] = (idx + 1) * (idx + 1);

    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 return type.
- *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. It that case the `Virtual` namespace would have been required for the function name and its return type, 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)
            (*ip)[idx] = (idx + 1) * (idx + 1);

        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 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[1].

Standard **C++** now offers a `string` type. 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 formally introduced 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 several 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

[1]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`

value is a (symbolic) value of type `string::size_type`, which is (for all practical purposes) an (`unsigned`) `int`.

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

Some `string`-members use *iterators*. Iterators will be covered in section 17.2. The member functions using 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>

using namespace std;

int main()
{
    string stringOne("Hello World");  // using plain ascii-Z
    string stringTwo(stringOne);      // using another string object
    string stringThree;               // implicit initialization to "". Do
                                      // not use the form '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>
using namespace std;

int main()
{
    string stringOne("Hello World");
    string 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 <iostream>
#include <string>
```

```
using namespace std;

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 accessed 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. If range checking is required the `string::at()` member function should be used:

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

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 Horld"
    if (stringOne.at(0) == 'H')
        stringOne.at(0) = 'W';  // now "Wello Horld"

    return 0;
}
```

  When an illegal index is passed to the `at()` member function, the program aborts (actually, an *exception* is generated, which could be caught. Exceptions are covered in chapter 8).

- 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");
    string 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::size_type pos, size_t n, string const &other)`: the first argument indicates the position in the current string that should be compared; the second argument indicates the number of characters that should be compared (if this value exceeds the number of characters that are actually available, only the available characters are compared); the third argument indicates the string which is compared to the current string.
- More variants of `string::compare()` are available. As stated, refer to section 4.2.4 for details.

The following example illustrates the `compare()` function:

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

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

```
        // comparing from a certain offset in stringOne
    if (!stringOne.compare(1, stringOne.length() - 1, "ello World"))
        cout << "comparing 'Hello world' from index 1"
                " to 'ello World': ok\n";

        // the number of characters to compare (2nd arg.)
        // may exceed the number of available characters:
    if (!stringOne.compare(1, string::npos, "ello World"))
        cout << "comparing 'Hello world' from index 1"
                " to 'ello World': ok\n";

        // comparing from a certain offset in stringOne over a
        // certain number of characters in "World and more"
        // This fails, as all of the chars in stringOne
        // starting at index 6 are compared, not just
        // 3 chars in "World and more"
    if (!stringOne.compare(6, 3, "World and more"))
        cout <<
        "comparing 'Hello World' from index 6 over"
        " 3 positions to 'World and more': ok\n";
    else
        cout << "Unequal (sub)strings\n";

        // This one will report a match, as only 5 characters are
        // compared of the  source and target strings
    if (!stringOne.compare(6, 5, "World and more", 0, 5))
        cout <<
        "comparing 'Hello World' from index 6 over"
        " 5 positions to 'World and more': ok\n";
    else
        cout << "Unequal (sub)strings\n";
}
/*
        Generated output:

    comparing 'Hello world' from index 1 to 'ello World': ok
    comparing 'Hello world' from index 1 to 'ello World': ok
    Unequal (sub)strings
    comparing 'Hello World' from index 6 over 5 positions to
                'World and more': ok
*/
```

- 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");
    string stringTwo("World");

    stringOne += " " + stringTwo;

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

    cout << stringOne << endl;

    string stringThree("Hello");
                                    // append " world":
    stringThree.append(stringOne, 5, 6);

    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 *file changer*: 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 applied as well.

```
    #include <iostream>
    #include <string>

    using namespace std;

    int main(int argc, char **argv)
    {
        if (argc == 3)
        {
            cerr << "Usage: <searchstring> <replacestring> to process "
                                                                "stdin\n";
            return 1;
        }

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

        if (search == replace)
        {
            cerr << "The replace and search texts should be different\n";
            return 1;
        }
```

```
    string line;
    while (getline(cin, line))
    {
        string::size_type idx = 0;
        while (true)
        {
            idx = line.find(search, idx); // find(): another string member
                                          //         see ‘searching’ below
            if (idx == string::npos)
                break;

            line.replace(idx, search.size(), replace);
            idx += replace.length();     // don’t change the replacement
        }
        cout << line << endl;
    }
    return 0;
}
```

- Swapping: The member function `string &string::swap(string &other)` swaps the contents of two `string`-objects. For example:

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

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

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

    stringOne.swap(stringTwo);

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

- 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 <iostream>
```

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

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

    stringOne.erase(5, 6);

    cout << stringOne << endl;

    stringOne.erase();

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

- 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 <iostream>
#include <string>
using namespace std;

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 starting at the first non-digit:

```
#include <iostream>
```

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

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"
        << "Starting at the first non-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 <iostream>
#include <string>
using namespace std;

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

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

```
}
```

- 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 <iostream>
#include <string>
using namespace std;

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";
}
```

- 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 line from a stream into a string: The function

```
istream &getline(istream &instream, string &target, char delimiter)
```

  may be used to read a line of text (up to the first delimiter or the end of the stream) from `instream` (note that `getline()` is not a *member* function of the class `string`).

  The delimiter has a default value `'\n'`. It is removed from `instream`, but it is *not* stored in `target`. The member `istream::eof()` 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).

- A `string` variables may be extracted from a stream. Using the construction

```
istr >> str;
```

  where `istr` is an `istream` object, and `str` is a `string`, the next consecutive series of non-blank characters will be assigned to `str`. Note that by default the extraction operation will skip any blanks that precede the characters that are extracted from the stream.

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 const &` 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. The entity referred to by `argument` always remains unchanged.

Furthermore, `opos` indicates an offset into the `object` string, `apos` indicates an offset into the `argument` string. Analogously, `on` indicates a number of characters in the `object` string, and `an` indicates a number of characters in the `argument` string. Both `opos` and `apos` must refer to existing offsets, or an exception will be generated. In contrast to this, `an` and `on` may exceed the number of available characters, in which case only the available characters will be considered.

When streams are involved, `istr` indicates a stream from which information is extracted, `ostr` indicates a stream into which information is inserted.

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*. Like `apos` and `opos`, iterators must also refer to an existing character, or to an available iterator range of the string to which they refer.

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 Initializers

The following `string` constructors are available:

- `string object`:

    Initializes `object` to an empty string.

- `string object(string::size_type no, char c)`:

    Initializes `object` with `no` characters `c`.

- `string object(string argument)`:

    Initializes `object` with `argument`.

- `string object = argument`:

    Initializes `object` with `argument`. This is an alternative form of the previous initialization.

- `string object(string argument, string::size_type apos, string::size_type an = pos)`:

    Initializes `object` with `argument`, using `an` characters of `argument`, starting at index `apos`.

- `string object(InputIterator begin, InputIterator end)`:

    Initializes `object` with the range of characters implied by the provided `InputIterators`. Iterators are covered in detail in section 17.2, but can (for the time being) be interpreted as pointers to characters. See also the next section.

4.2.2 Iterators

See section 17.2 for details about *iterators*. As a quick introduction to iterators: an iterator acts like a pointer, and pointers can often be used in situations where iterators are requested. Iterators almost always come in pairs: the begin-iterator points to the first entity that will be considered, the end-iterator points just beyond the last entity that will be considered. Iterators play an important role in the context of *generic algorithms* (cf. chapter 17).

- Forward iterators are returned by the members:
  - `string::begin()`, pointing to the first character inside the string object.
  - `string::end()`, pointing beyond the last character inside the string object.
- Reverse iterators are also iterators, but they are used to step through a range in a reversed direction. Reverse iterators are returned by the members:
  - `string::rbegin()`, which can be considered to be an iterator pointing to the last character inside the string object.
  - `string::rend()`, which can be considered to be an iterator pointing before the first character inside the string object.

4.2.3 Operators

The following string operators are available:

- `object = argument`.

  Assignment of `argument` to an existing string `object`.

- `object = c`.

  Assignment of `char c` to `object`.

- `object += argument`.

  Appends `argument` to `object`. `Argument` may also be a `char` expression.

- `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` expression, as illustrated by the following example:

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

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

- object[string::size_type opos].

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

- argument1 == argument2.

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

- ostr << object.

  The insertion-operator may be used with string objects.

- istr >> 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 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 unsigned.

The special value string::npos, defined by the string class, represents a non-existing index. This value is returned by all members returning indices when they could not perform their requested tasks. Note that the string’s *length* is not returned as a valid index. E.g., when calling a member ‘find_first_not_of(" ")’ (see below) on a string object holding 10 blank space characters, npos is returned, as the string only contains blanks. The final 0-byte that is used in **C** to indicate the end of a ASCII-Z string is *not* considered part of a **C++** string, and so the member function will return npos, rather than length().

In the following overview, ‘size_type’ should always be read as ‘string::size_type’.

- char &string::at(size_type opos):

  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 apos, size_type an)`:
  - If only `argument` is provided, it is appended to the `string` object.
  - If `apos` is provided as well, `argument` is appended from index position `apos` until the end of `argument`.
  - If `an` is provided too, `an` characters of `argument`, starting at index position `apos` are appended to the `string` object.

  If `argument` is of type `char const *`, the second parameter `apos` is not available. 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. Of course, if `apos` and `an` *are* specified in this case, `append()` can still be used: the `char const *` argument will then implicitly be converted to a `string const &`.

- `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 apos, size_type an)`:
  - If only `argument` is provided, it is assigned to the `string` object.
  - If `apos` is specified as well, a substring of `argument` object, starting at offset position `apos`, is assigned to the `string` object calling this member.
  - If `an` is provided too, a substring of `argument` object, starting at offset position `apos`, containing at most `an` characters, is assigned to the `string` object calling this member.

  If `argument` is of type `char const *`, no parameter `apos` 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. As with the `string::append()` member, a `char const *` argument *may* be used, but it will be converted to a `string` object first.

- `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 opos, size_type on, 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 `on` characters, starting at offset `opos`, are compared with the text in `argument`. The `argument` may also be a (non-0) `char const *`.

- `int string::compare(size_type opos, size_type on, string argument, size_type apos, size_type an)`:

  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 `on` characters of the `string` object, starting at offset `opos`, are compared with at most `an` characters of `argument`, starting at offset `apos`. Note that `argument` *must* also be a `string` object.

- `int string::compare(size_type opos, size_type on, char const *argument, size_type an)`:

  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 `on` characters of the `string` object, starting at offset `opos`, are compared with at most `an` characters of `argument`. `Argument` must have at least `an` characters. However, the characters may have arbitrary values: the ASCII-Z value has no special meaning.

- `size_type string::copy(char *argument, size_type on, size_type opos)`:

  The contents of the `string` object is (partially) copied to `argument`.

  - If `on` is provided, it refers to the maximum number of characters that will be copied. If omitted, all the `string`'s characters, starting at offset `opos`, will be copied to `argument`. Also, `string::npos` may be specified to indicate that all available characters should be copied.
  - If both `on` and `opos` are provided, `opos` refers to the offset in the `string` object where copying should start.

  The actual number of characters that were copied is returned. Note: *following the copying, no `ASCII-Z` will be appended to the copied string*. A final ASCII-Z character can be appended to the copied text using the following construction:

  ```
  buffer[s.copy(buffer)] = 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. Since this member does not return an ascii-Z string (as `c_str()` does), it can be used to store and retrieve any kind of information, including, e.g., series of 0-bytes:

  ```
  string s;
  s.resize(2);
  cout << static_cast<int>(s.data()[1]) << endl;
  ```

- `bool string::empty()`:

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

- `string &string::erase(size_type opos; size_type on)`:

  This member function can be used to erase (a sub)string of the `string` object.

  - If no arguments are provided, the contents of the `string` object are completely erased.
  - If `opos` is specified, the contents of the `string` object are erased, starting from index position `opos` until (including) the object's final character.

  - If `on` is provided as well, `on` characters of the `string` object, starting at index position `opos` are erased.

- `iterator string::erase(iterator obegin, iterator oend)`:

  - If only `obegin` is provided, the `string` object's character at iterator position `obegin` is erased.
  - If `oend` is provided as well, the range of characters of the `string` object, implied by the `iterators obegin` and `oend` are erased.

  The iterator `obegin` is returned, pointing to the character immediately following the last erased character.

- `size_type string::find(string argument, size_type opos)`:

  Returns the index in the `string` object where `argument` is found.

  - If `opos` is provided, it refers to the index in the `string` object where the search for `argument` should start. If `opos` is omitted, searching starts at the beginning of the `string` object.

- `size_type string::find(char const *argument, size_type opos, size_type an)`:

  Returns the index in the `string` object where `argument` is found.

  - If `opos` is provided, it refers to the index in the `string` object where the search for `argument` should start. If omitted, the `string` object is scanned completely.
  - If `an` is provided as well, it 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.

- `size_type string::find(char c, size_type opos)`:

  Returns the index in the `string` object where `c` is found.

  - If `opos` is provided it refers to the index in the `string` object where the search for the character should start. If omitted, searching starts at the beginning of the `string` object.

- `size_type string::find_first_of(string argument, size_type opos)`:

  Returns the index in the `string` object where any character in `argument` is found.

  - If `opos` is provided, it refers to the index in the `string` object where the search for `argument` should start. If omitted, searching starts at the beginning of the `string` object.

- `size_type string::find_first_of(char const *argument, size_type opos, size_type an)`:

  Returns the index in the `string` object where a character of `argument` is found, no matter which character.

  - If `opos` is provided it refers to the index in the `string` object where the search for `argument` should start. If omitted, the `string` object is scanned completely.
  - If `an` is provided it indicates the number of characters of the `char const *` argument that should be used in the search: it defines a partial string starting at the beginning of the `char const *` argument. If omitted, all of `argument`'s characters are used.

- `size_type string::find_first_of(char c, size_type opos)`:

  Returns the index in the `string` object where character `c` is found.

  - If `opos` is provided, it refers to the index in the `string` object where the search for `c` should start. If omitted, searching starts at the beginning of the `string` object.

- `size_type string::find_first_not_of(string argument, size_type opos)`:

  Returns the index in the `string` object where a character not appearing in `argument` is found.

  - If `opos` is provided, it refers to the index in the `string` object where the search for `argument` should start. If omitted, searching starts at the beginning of the `string` object.

- `size_type string::find_first_not_of(char const *argument, size_type opos, size_type an)`:

  Returns the index in the `string` object where any character *not* appearing in `argument` is found.

  - If `opos` is provided it refers to the index in the `string` object where the search for characters not specified in `argument` should start. If omitted, the `string` object is scanned completely.
  - If `an` is provided it indicates the number of characters of the `char const *` argument that should be used in the search: it defines a partial string starting at the beginning of the `char const *` argument. If omitted, all of `argument`'s characters are used.

- `size_type string::find_first_not_of(char c, size_type opos)`:

  Returns the index in the `string` object where another character than `c` is found.

  - If `opos` is provided, it refers to the index in the `string` object where the search for `c` should start. If omitted, searching starts at the beginning of the `string` object.

- `size_type string::find_last_of(string argument, size_type opos)`:

  Returns the *last* index in the `string` object where one of `argument`'s characters is found.

  - If `opos` is provided it refers to the index in the `string` object where the search for `argument` should start, proceeding backwards to the `string`'s first character. If omitted, searching starts at the the `string` object's last character.

- `size_type string::find_last_of(char const* argument, size_type opos, size_type an)`:

  Returns the last index in the `string` object where one of `argument`'s characters is found.

  - If `opos` is provided it refers to the index in the `string` object where the search for `argument` should start, proceeding backwards to the `string`'s first character. If omitted, searching starts at the the `string` object's last character.
  - If `an` is provided it indicates the number of characters of `argument` that should be used in the search: it defines a partial string starting at the beginning of the `char const *` argument. If omitted, all of `argument`'s characters are used.

- `size_type string::find_last_of(char c, size_type opos)`:

  Returns the last index in the `string` object where character `c` is found.

  - If `opos` is provided it refers to the index in the `string` object where the search for character `c` should start, proceeding backwards to the `string`'s first character. If omitted, searching starts at the the `string` object's last character.

- `size_type string::find_last_not_of(string argument, size_type opos)`:

  Returns the last index in the `string` object where any character not appearing in `argument` is found.

  - If `opos` is provided it refers to the index in the `string` object where the search for characters not appearing in `argument` should start, proceeding backwards to the `string`'s first character. If omitted, searching starts at the the `string` object's last character.

- `size_type string::find_last_not_of(char const *argument, size_type opos, size_type an)`:

  Returns the last index in the `string` object where any character not appearing in `argument` is found.

  - If `opos` is provided it refers to the index in the `string` object where the search for characters not appearing in `argument` should start, proceeding backwards to the `string`'s first character. If omitted, searching starts at the the `string` object's last character.
  - If `an` is provided it indicates the number of characters of `argument` that should be used in the search: it defines a partial string starting at the beginning of the `char const *` argument. If omitted, all of `argument`'s characters are used.

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

  Returns the last index in the `string` object where another character than `c` is found.

  - If `opos` is provided it refers to the index in the `string` object where the search for a character unequal to character `c` should start, proceeding backwards to the `string`'s first character. If omitted, searching starts at the the `string` object's last character.

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

  This function (note that it's not a *member* function of the class `string`) can be used to read a line of text from `istr`. All characters until `delimiter` (or the end of the stream, whichever comes first) are read from `istr` and are stored in `object`. The delimiter, when present, is removed from the stream, but is not stored in `line`. The delimiter's default value is `'\n'`.
  If the delimiter is not found, `istr.fail()` returns 1 (see section 5.3.1). Note that the contents of the last line, whether or not it was terminated by a delimiter, will always be assigned to `object`.

- `string &string::insert(size_type opos, string argument, size_type apos, size_type an)`:

  This member function can be used to insert (a sub)string of `argument` into the `string` object, *at* the `string` object's index position `opos`. The arguments `apos` and `an` must either be specified or they must both be omitted. If specified, `an` characters of `argument`, starting at index position `apos` are inserted into the `string` object.
  If `argument` is of type `char const *`, no parameter `apos` is available. So, with

`char const *` arguments, either *all* characters or an *initial subset* of `an` characters of the provided `char const *` argument are inserted into the `string` object. In this case, the prototype of the member function is:

```
string &string::insert(size_type opos, char const *argument,
                       size_type an)
```

(As before, an implicit conversion from `char const *` to `string` will occur if `apos` and `an` *are* provided).

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

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

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

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

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

  At the (iterator) position `obegin` of `object` `n` characters `c` are inserted. The iterator `obegin` is returned.

- `iterator string::insert(iterator obegin, InputIterator abegin, InputIterator aend)`:

  The range of characters implied by the `InputIterators abegin` and `aend` are inserted at the (iterator) position `obegin` in `object`. The iterator `obegin` 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 opos, size_type on, string argument, size_type apos, size_type an)`:

  The arguments `apos` and `an` are optional. If omitted, `argument` is considered completely. The substring of `on` characters of the `string` object, starting at position `opos` is replaced by `argument`. If `on` is set to 0, the member function *inserts* `argument` into `object`.

  - If `apos` and `an` are provided, `an` characters of `argument`, starting at index position `apos` will replace the indicated range of characters of `object`.

  If `argument` is of type `char const *`, no parameter `apos` is available. So, with `char const *` arguments, either *all* characters or an *initial subset* of the characters of `an` characters of the provided `char const *` argument will replace the indicated range of characters in `object`. In that case, the prototype of the member function is:

```
string &string::replace(size_type opos, size_type on,
                        char const *argument, size_type an)
```

- `string &string::replace(size_type opos, size_type on, size_type n, char c)`:

  This member function can be used to replace `on` characters of the `string` object, starting at index position `opos`, by `n` characters having values `c`.

- `string &string::replace (iterator obegin, iterator oend, string argument)`:

    Here, the string implied by the iterators `obegin` and `oend` 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.

- `string &string::replace(iterator obegin, iterator oend, size_type n, char c)`:

    The range of characters of the `string` object, implied by the `iterators obegin` and `oend` are replaced by `n` characters having values `c`.

- `string string::replace(iterator obegin, iterator oend, InputIterator abegin, InputIterator aend)`:

    Here the range of characters implied by the iterators `obegin` and `oend` is replaced by the range of characters implied by the `InputIterators abegin` and `aend`.

- `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, in which case the value `c = 0` is used. If provided and the string is enlarged, the extra characters are initialized to `c`.

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

    Returns the index in the `string` object where `argument` is found. Searching proceeds either from the end of the `string` object or from its offset `opos` back to the beginning. If the argument `opos` is omitted, searching starts at the end of `object`.

- `size_type string::rfind(char const *argument, size_type opos, size_type an)`:

    Returns the index in the `string` object where `argument` is found. Searching proceeds either from the end of the `string` object or from offset `opos` back to the beginning. The parameter `an` 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.

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

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

- `size_type string::size()`:

    returns the number of characters stored in the `string` object. This member is a synonym of `string::length()`.

- `string string::substr(size_type opos, size_type on)`:

    Returns (using a *value* return type) a substring of the `string` object. The parameter `on` may be used to specify the number of characters of `object` that are returned. The parameter `opos` may be used to specify the index of the first character of `object` that is returned. Either `on` or both arguments may be omitted. The string `object` itself is not modified by `substr()`.

- `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 *`. Of course, both strings (`object` and `argument`) are modified by this member function.

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 (<<) and the extraction 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* will formally be introduced in chapter 13, we think it is well possible to introduce input and output (I/O) facilities long before the technical background of these topics is actually covered.

Most **C++** I/O classes have names starting with `basic_` (like `basic_ios`). However, these `basic_` names are not regularly found in **C++** programs, as most classes are also defined using `typedef` definitions like:

```
        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 thereupon be used for both the `char` and `wchar_t` types. So, analogously to the above `typedef` there exists a

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

This type definition can be used for the `wchar_t` type. Because of the existence of these type definitions, the `basic_` prefix can be omitted from the Annotations without loss of continuity. 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 20, 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 with 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.
- Basic **C++** input facilities are available in the `istream` class. This class defines the insertion operator and related facilities for input. Analogous to the `ostringstream` a class `istringstream` class is available for extracting information from memory buffers.

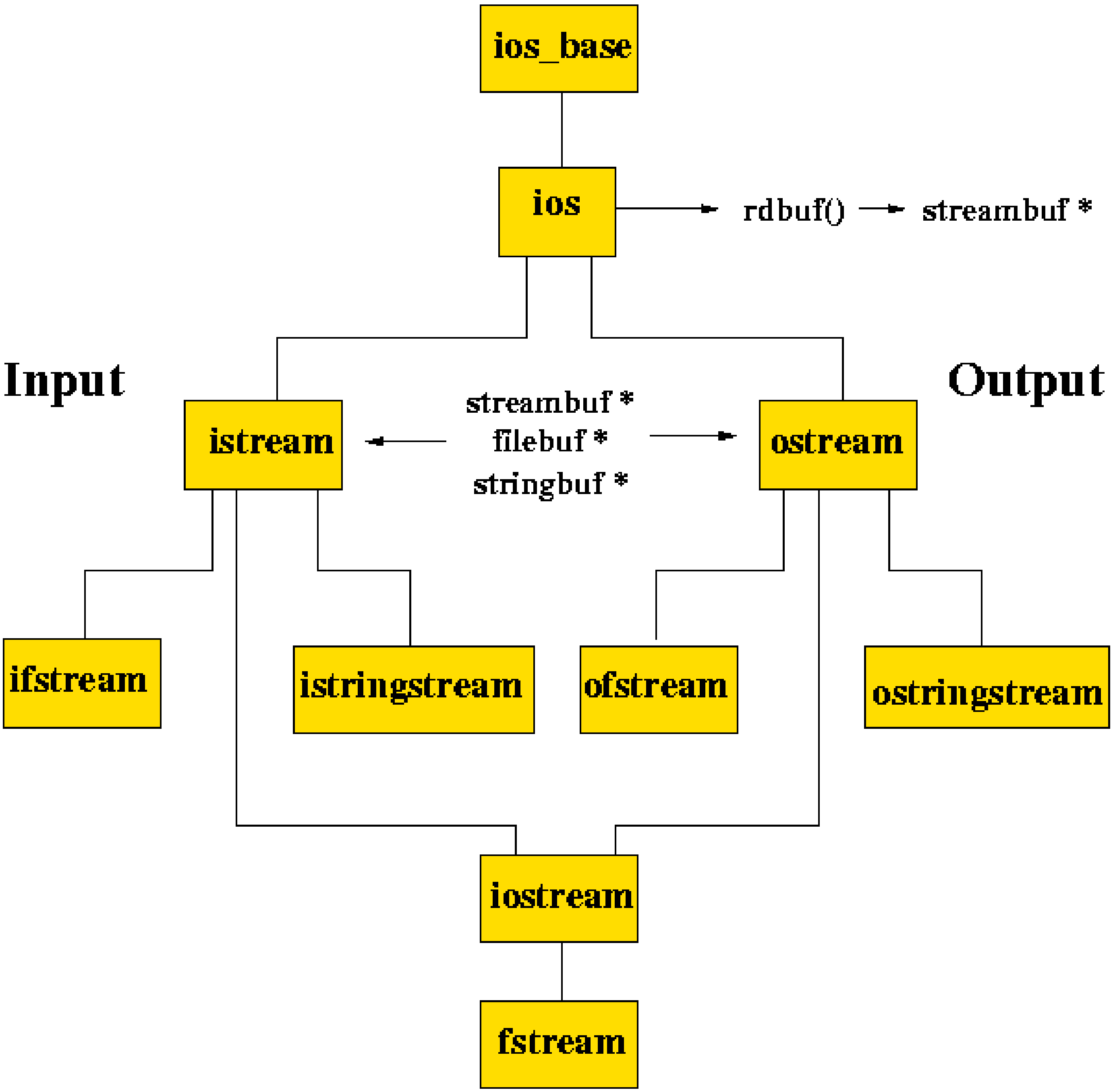

Figure 5.1: Central I/O Classes

- 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 ojects. Other I/O related topics are covered elsewhere in the Annotations, e.g., in chapter 20.

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 Special 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 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 foundation 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 its derivatives.

For the sake of completeness it is noted 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, facilities actually provided by `ios_base` will be discussed as facilities provided by `ios`. The reader who is interested in the true class in which a particular facility is defined should consult the relevant header files (e.g., `ios_base.h` and `basic_ios.h`).

5.3 Interfacing 'streambuf' objects: the class 'ios'

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

Although it *is* possible to construct an `ios` object directly, this is hardly ever done. The purpose of the class `ios` is to provide the facilities of the class `basic_ios`, and to add several new facilites, 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 other stream classes. Before discussing these additional stream classes, the facilities offered by the class `ios` (and by implication: by `ios_base`) are now introduced.

The class `ios` 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 20.1.2 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 (see the next member). 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 requested at the level of the `streambuf` object to which the stream interfaces. See the member functions below for some examples.

- `ios::eofbit`:

  if this flag has been raised, the `ios` object has sensed end of file.

- `ios::failbit`:

  if this flag has been raised, an operation performed by the stream object has failed (like an attempt to extract an `int` when no numeric characters are available on input). In this case the stream itself could not perform the operation that was requested of it.

- `ios::goodbit`:

  this flag is raised when none of the other three condition flags were raised.

Several condition member functions are available to manipulate or determine the states of `ios` objects. Originally they returned `int` values, but their current return type is `bool`:

- `ios::bad()`:

  this member function returns `true` when `ios::badbit` has been set and `false` otherwise. If `true` is returned it indicates that an illegal operation has been requested at the level of the `streambuf` object to which the stream interfaces. What does this mean? It indicates that the `streambuf` itself is behaving unexpectedly. Consider the following example:

  ```
  std::ostream error(0);
  ```

This constructs an `ostream` object *without* providing it with a working `streambuf` object. Since this '`streambuf`' will never operate properly, its `ios::badbit` is raised from the very beginning: `error.bad()` returns `true`.

- `ios::eof()`:

  this member function returns `true` when end of file (`EOF`) has been sensed (i.e., `ios::eofbit` has been set) and `false` otherwise. Assume we're reading lines line-by-line from `cin`, but the last line is not terminated by a final `\n` character. In that case `getline()`, attempting to read the `\n` delimiter, hits end-of-file first. This sets `eos::eofbit`, and `cin.eof()` returns `true`. For example, assume `main()` executes the statements:

  ```
  getline(cin, str);
  cout << cin.eof();
  ```

  Following:

  ```
  echo "hello world" | program
  ```

  the value 0 (no EOF sensed) is printed, following:

  ```
  echo -n "hello world" | program
  ```

  the value 1 (EOF sensed) is printed.

- `ios::fail()`:

  this member function returns `true` when ios::bad() returns `true` or when the ios::failbit was set, and `false` otherwise. In the above example, `cin.fail()` returns `false`, whether we terminate the final line with a delimiter or not (as we've read a line). However, trying to execute a second `getline()` statement will set `ios::failbit`, causing `cin::fail()` to return `true`. The value `not fail()` is returned by the `bool` interpretation of a stream object (see below).

- `ios::good()`:

  this member function returns the value of the `ios::goodbit` flag. It returns `true` when none of the other condition flags (`ios::badbit, ios::eofbit, ios::failbit`) were raised. Consider the following little program:

  ```
  #include <iostream>
  #include <string>

  using namespace std;

  void state()
  {
      cout << "\n"
              "Bad: " << cin.bad() << " "
              "Fail: " << cin.fail() << " "
              "Eof: " << cin.eof() << " "
              "Good: " << cin.good() << endl;
  }

  int main()
  {
      string line;
      int x;
  ```

```
        cin >> x;
        state();

        cin.clear();
        getline(cin, line);
        state();

        getline(cin, line);
        state();
    }
```

When this program processes a file having two lines, containing, respectively, `hello` and `world`, while the second line is not terminated by a `\n` character it shows the following results:

```
Bad: 0 Fail: 1 Eof: 0 Good: 0

Bad: 0 Fail: 0 Eof: 0 Good: 1

Bad: 0 Fail: 0 Eof: 1 Good: 0
```

So, extracting `x` fails (`good()` returning `false`). Then, the error state is cleared, and the first line is successfully read (`good()` returning `true`). Finally the second line is read (incompletely): `good()` returns t(false), and `eof()` returns `true`.

- Interpreting streams as `bool` values:

  streams may be used in expressions expecting logical values. Some examples are:

  ```
  if (cin)                // cin itself interpreted as bool
  if (cin >> x)           // cin interpreted as bool after an extraction
  if (getline(cin, str))  // getline returning cin
  ```

  When interpreting a stream as a logical value, it is actually `not ios::fail()` that is interpreted. So, the above examples may be rewritten as:

  ```
  if (not cin.fail())
  if (not (cin >> x).fail())
  if (not getline(cin, str).fail())
  ```

  The former incantation, however, is used almost exclusively.

The following members are available to manage 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)`. It's return type is `void`

- `ios::rdstate()`:

  This member function returns (as an `int`) 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. Its return type is `void`. 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 is used when it is necessary to control the width of an output field or an input buffer and if formatting is used to determine the form (e.g., the *radix*) in which a value is displayed. Most formatting belongs to the realm of the `ios` class, although most formatting is actually used with output streams, like the upcoming `ostream` class. Since the formatting is controlled by flags, 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.

Formatting is controlled by a set of *formatting flags*. These flags can basically be altered in two ways: using specialized member functions, discussed in section 5.3.2.2 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).

5.3.2.1 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 5.3.2.2 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`). Example, setting the value 10 left-aligned in a field of 10 character positions:

  ```
  cout.setf(ios::left, ios::adjustfield);
  cout << "'" << setw(10) << 10 << "'" << endl;
  ```

- `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`). Example, printing the value 57005 as a hexadecimal number:

  ```
  cout.setf(ios::hex, ios::basefield);
  cout << 57005 << endl;
      // or, using the manipulator:
  cout << hex << 57005 << endl;
  ```

- `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: `boolalpha` and `noboolalpha`. Example, printing the boolean value 'true' instead of 1:

  ```
  cout << boolalpha << (1 == 1) << endl;
  ```

- `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: `dec`.

- `ios::fixed`:

  to display real values in a fixed notation (e.g., 12.25), as opposed to displaying values in a scientific notation. If just a change of notation is requested the mask value `ios::floatfield` must be provided when `setf()` is used. Example: see `ios::scientific` below. Corresponding manipulator: `fixed`.

  Another use of `ios::fixed` is to set a fixed number of digits behind the decimal point when floating or double values are to be printed. See `ios::precision` in section 5.3.2.2.

- `ios::floatfield`:

  mask value used in combination with a flag setting the way real numbers are displayed (`ios::fixed` or `ios::scientific`). Example:

  ```
  cout.setf(ios::fixed, ios::floatfield);
  ```

- `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: `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: `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: `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: `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: `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: `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: `showbase` and `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: `showpoint`.

- `ios::showpos`:

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

- `ios::skipws`:

  used for extracting information from streams. When this flag is set (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.

5.3.2.2 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. The current `ios` object is returned.

- `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: `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. Note: when setting flags using this member, a previously set flag may have to be unset first. For example, to change the number conversion of `cout` from decimal to hexadecimal using this member, do:

  ```
  cout.flags(ios::hex | cout.flags() & ~ios::dec);
  ```

  Alternatively, either of the following statements could have been used:

  ```
  cout.setf(ios::hex, ios::basefield);
  cout << hex;
  ```

- `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: `setprecision()`. Example, rounding all displayed double values to a fixed number of digits (e.g., 3) behind the decimal point:

  ```
  cout.setf(ios::fixed);
  cout.precision(3);
  cout << 3.0 << " " << 3.01 << " " << 3.001 << endl;
  cout << 3.0004 << " " << 3.0005 << " " << 3.0006 << endl;
  ```

Note that the value 3.0005 is rounded away from zero to 3.001 (-3.0005 is rounded to -3.001).

- `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: `setiosflags` and `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: `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. Corresponding manipulator: `setw(int)`.

5.4 Output

In **C++** output is primarily based on the `ostream` class. The `ostream` class defines the basic operators and members for inserting information into streams: the *insertion 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).

We can construct `ostream` objects using the following *ostream constructor*:

- `ostream object(streambuf *sb)`:

  this constructor can be used to construct a wrapper around an existing `streambuf`, which may be the interface to an existing file. See chapter 20 for examples.

  What this boils down to is that it isn't possible to construct a plain `ostream` object that can be used for insertions. When `cout` or its friends is used, we are actually using a predefined `ostream` object that has already been created for us, and interfaces to, e.g., the standard output stream using a (also predefined) `streambuf` object handling the actual interfacing.

  Note that it *is* possible to construct an `ostream` object passing it a ih(std::ostream: constructed using a 0-pointer) 0-pointer as a `streambuf`. Such an object cannot be used for insertions (i.e., it will raise its `ios::bad` flag when something is inserted into it), but since it may be given a `streambuf` later, it may be preliminary constructed, receiving its `streambuf` once it becomes available.

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.

5.4.1.1 Writing to 'ostream' objects

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

The *insertion 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 << 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 processed.

Streams do not have facilities for formatted output like **C**'s `form()` and `vform()` functions. Although it is not difficult to realize these facilities in the world of streams, `form()`-like functionality is hardly ever required in **C++** programs. Furthermore, as it is potentially type-*unsafe*, it might be better to avoid this functionality completely.

When binary files must be written, normally no text-formatting is used or required: an `int` value should be written as a series of unaltered bytes, not as a series of ASCII numeric characters 0 to 9. The following member functions of `ostream` objects may be used to write 'binary files':

- `ostream& 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. Note that the first argument is a `char const *`: a *type_cast* is required to write any other type. For example, to write an `int` as an unformatted series of byte-values:

  ```
  int x;
  out.write(reinterpret_cast<char const *>(&x), sizeof(int));
  ```

5.4.1.2 'ostream' positioning

Although not every `ostream` object supports repositioning, they usually do. This means that it is possible to rewrite a section of the stream which was written earlier. Repositioning is frequently used in *database applications* where it must be possible to access the information in the database 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 the stream.

  It is ok 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.

5.4.1.3 'ostream' flushing

Unless the `ios::unitbuf` flag has been set, information written to an `ostream` object is not immediately written to the physical stream. Rather, an internal buffer is filled 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 5.4.2.1 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)`:

  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
*/
```

5.4.2.1 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.
  Note that the original contents are *only* kept if some other flag tells the object to do so. For example `ofstream out("gone", ios::ate)` will *rewrite* the file `gone`, because the implied `ios::out` will cause the rewriting. If rewriting of an existing file should be prevented, the `ios::in` mode should be specified too. Note that in this case the construction only succeeds if the file already exists.

- `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:

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

- `ostringstream::str(string)`:

  This member function will re-initialize the `ostringstream` object with new initial contents.

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`.

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. Such `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.

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

In **C++** input is primarily based on the `istream` class. The `istream` class defines the basic operators and members for extracting information from streams: the *extraction 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. Similarly to `ostream` objects, `istream` objects may ih(std::istream: constructed using a 0-pointer) initially be constructed using a 0-pointer. See section 5.4.1 for a discussion, and chapter 20 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.

5.5.1.1 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 >> 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 >> 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 input (like **C**'s `scanf()` and `vscanf()` functions). Although it is not difficult to make these facilities available in the world of streams, `scanf()`-like functionality is hardly ever required in **C++** programs. Furthermore, as it is potentially type-*unsafe*, it might be better to avoid this functionality completely.

When binary files must be 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. Note that the `std::string` class also has a support function `getline()` which is used more often than this `istream::getline()` member function (see section 4.2.4).

- `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()`

5.5.1.2 'istream' positioning

Although not every `istream` object supports repositioning, some do. This means that it is possible to read the same section of a stream repeatedly. Repositioning is frequently used in *database applications* where it must be possible to access the information in the database randomly.

The following members are available:

- `pos_type istream::tellg()`:

  this function returns the current (absolute) position where the next read-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 an `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 ok 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 5.4.2.1 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.

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

  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 5.5.1.1), 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 extracted 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 5.3.2.1). The format flags can be controlled by member functions (see section 5.3.2.2), 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 9.10.1. In this section the manipulators that are available in the **C++** I/O library are discussed. Most manipulators affect *format flags*. See section 5.3.2.1 for details about these flags. Most manipulators are parameterless. Sources in which manipulators expecting arguments are used, must do:

```
#include <iomanip>
```

- `std::boolalpha`:

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

- `std::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 `std::hex` and `std::oct`, below):

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

- `std::endl`:

  This manipulator will insert a newline character into an output buffer and will flush the buffer thereafter.

- `std::ends`:

  This manipulator will insert a string termination character into an output buffer.

- `std::fixed`:

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

- `std::flush`:

  This manipulator will flush an output buffer.

- `std::hex`:

  This manipulator enforces the display and reading of integral numbers in hexadecimal format.

- `std::internal`:

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

- `std::left`:

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

- `std::noboolalpha`:

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

- `std::noshowpoint`:

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

- `std::noshowpos`:

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

- `std::noshowbase`:

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

- `std::noskipws`:

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

- `std::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.

- `std::nouppercase`:

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

- `std::oct`:

  This manipulator enforces the display and reading of integral numbers in octal format.

- `std::resetiosflags(flags)`:

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

- `std::right`:

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

- `std::scientific`:

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

- `std::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.

- `std::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.

- `std::setiosflags(flags)`:

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

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

  This manipulator will set the precision in which a `float` or `double` is displayed. In combination with `std::fixed` it can be used to display a fixed number of digits of the fractional part of a floating or double value:

  ```
  cout << fixed << setprecision(3) << 5.0 << endl;
  // displays: 5.000
  ```

- `std::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....

- `std::showbase`:

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

- `std::showpoint`:

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

- `std::showpos`:

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

- `std::skipws`:

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

- `std::unitbuf`:

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

- `std::uppercase`:

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

- `std::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 plays 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`. However, this section of the current chapter assumes a basic familiarity with the concept of polymorphism, a topic discussed in detail in chapter 14. Readers not yet familiar with the concept of polymorphism may, for the time being, skip this section without loss of continuity.

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, encountered also in, e.g., the TCP/IP stack. A `streambuf` can be considered yet another example of the chain of command pattern: here the program talks to `stream` objects, which in turn forward their requests to `streambuf` objects, which in turn communicate with the devices. Thus, as we will see shortly, we are 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 20 several other implications of using streambuf objects are mentioned. In the current chapter examples of copying streams, of redirecting streams and 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 streambuf object. Caution should be exercised when using this function: often there is a maximum of just one character that can be put back.

- int 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 there is a maximum of just one character that 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 `streambuf` object. If, after writing the character, the buffer is full, the function calls the (protected) member function `overflow()` to flush the buffer to the device (see section 5.7.1 below).

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

  This member function inserts `n` characters from `buffer` into the `streambuf` 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 normally not accessible. However, they are accessible in specializing classes which are derived from `streambuf`. 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, directly using its data members 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 begin-, actual- and end-pointers have been defined, as depicted in figure 5.2. In upcoming sections we will refer to this figure repeatedly.

Protected constructor:

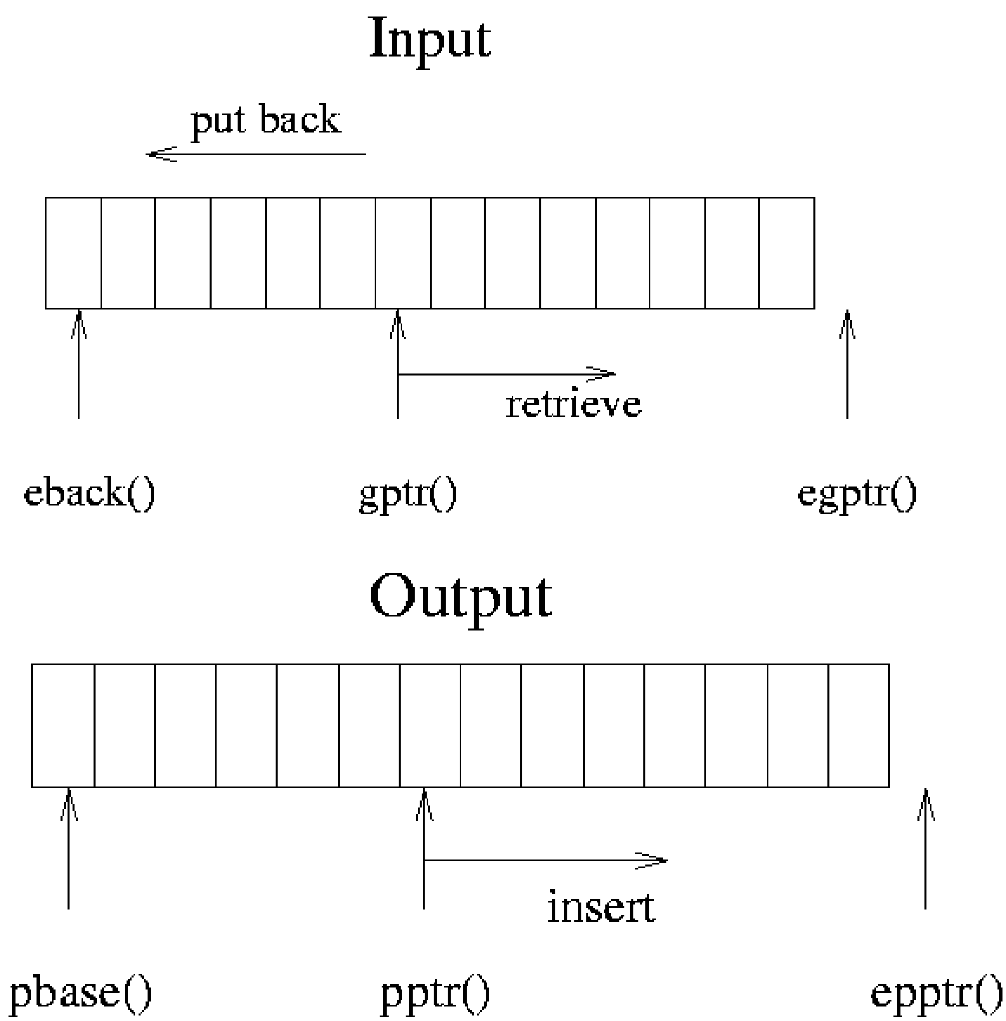

Figure 5.2: Input- and output buffer pointers of the class ‘streambuf’

- `streambuf::streambuf()`:

  Default (protected) constructor of the `class streambuf`.

Several protected member functions are related to input operations. The member functions marked as `virtual` may be redefined in classes derived from `streambuf`. In those cases, the redefined function will be called by `i/ostream` objects that received the addresses of such derived class objects. See chapter 14 for details about virtual member functions. Here are the protected members:

- `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 should be realized by calling `underflow()`, see below.

- `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.

- `virtual 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. Ususally `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 (not *no* arguments, but arguments having 0 values.) See also the member `streambuf::uflow()`, below.

- `virtual 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`).

- `virtual 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()`. When no input buffering is required this function, rather than `underflow()` can be overridden to produce the next available character from the device to read.

- `virtual 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*.

- `virtual 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.

Here are the protected member functions related to output operations. Similarly to the functions related to input operations, some of the following functions are `virtual`: they may be redefined in derived classes:

- `virtual 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. It receives as parameter `c` the next character to be processed by the `streambuf`. 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::epptr()`:

  For the output buffer the `class streambuf` maintains three pointers: `epptr()` points just beyond the location of the last character that can be written. See also figure 5.2. If `pptr()` (see below) equals `epptr()` 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 actual number of inserted characters should be returned. 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:

- `virtual 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.

- `virtual 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 (using `ios::beg`, `ios::cur` or `ios::end`). The default implementation is to indicate failure by returning -1. The function is called when, e.g., `tellg()` or `tellp()` is called. When a `streambuf` specialization supports seeking, then the specialization should also define this function to determine what to do with a repositioning (or `tellp/g()`) request.

- `virtual 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 (i.e, relative to `ios::beg`). The default implementation is to indicate failure by returning -1.

- `virtual 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 successfull 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 **C++** Annotations (e.g., in chapter 20).

Objects of the class `fstream` use a combined input/output buffer. This results from the fact that `istream` and `ostream`, are virtually derived from `ios`, which contains the `streambuf`. As explained in section 14.4.2, this implies that classes derived from both `istream` and `ostream` share their streambuf pointer. In order to construct a class supporting both input and output on separate buffers, the streambuf itself may define internally two buffers. When seekoff() is called for reading, its mode parameter is set to ios::in, otherwise to ios::out. This way, the streambuf specializaiton knows whether it should access the read buffer or the write buffer. Of course, underflow() and overflow() themselves already know on which buffer they should operate.

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.

Before filebuf objects can be defined the following preprocessor directive must have been specified:

```
#include <fstream>
```

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 5.5.1.1) 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 uses 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 all buffered output of the `ostream` object (by calling `flush()`) whenever an input or output operation is performed on the `ios` object to which the `ostream` object is tied. By default `cout` is tied to `cin` (i.e., `cin.tie(cout)`): whenever an operation on `cin` is requested, `cout` is flushed first. To break the coupling, the member function `ios::tie(0)` can be called.

Another (frequently useful, but non-default) 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 ";
    cerr << "first (unbuffered) line to cerr\n";
    cout << "\n";

    cerr.tie(&cout);

    cout << "second (buffered) line to cout ";
    cerr << "second (unbuffered) line to cerr\n";
    cout << "\n";
}
/*
    Generated output:

first (buffered) line to cout
first (unbuffered) line to cerr
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 Redirecting 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 20.3.1).

A standard situation where redirection is wanted is to write error messages to file rather than to standard error, 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 <streambuf>
#include <fstream>

using namespace std;

int main(int argc, char **argv)
{
    ofstream errlog;                                // 1
    streambuf *cerr_buffer = 0;                     // 2

    if (argc == 2)
    {
        errlog.open(argv[1]);                       // 3
        cerr_buffer = cerr.rdbuf(errlog.rdbuf());   // 4
    }
    else
    {
        cerr << "Missing log filename\n";
        return 1;
    }

    cerr << "Several messages to stderr, msg 1\n";
    cerr << "Several messages to stderr, msg 2\n";

    cout << "Now inspect the contents of " <<
            argv[1] << "... [Enter] ";
    cin.get();                                      // 5

    cerr << "Several messages to stderr, msg 3\n";

    cerr.rdbuf(cerr_buffer);                        // 6
    cerr << "Done\n";                               // 7
}
/*
    Generated output on file argv[1]

    at cin.get():

Several messages to stderr, msg 1
Several messages to stderr, msg 2
```

```
    at the end of the program:

Several messages to stderr, msg 1
Several messages to stderr, msg 2
Several messages to stderr, msg 3
*/
```

- At lines 1-2 local variables are defined: `errlog` is the `ofstream` to write the error messages too, and `cerr_buffer` is a pointer to a `streambuf`, to point to the original `cerr` buffer. This is further discussed below.
- At line 3 the alternate error stream is opened.
- At line 4 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 5 we pause. At this point, two lines were written to the alternate error file. We get a chance to take a look at its contents: there were indeed two lines written to the file.
- At line 6 the redirection is terminated. This is very important, as the `errlog` object is destroyed at the end of `main()`. If `cerr`'s buffer would not have been restored, then at that point `cerr` would refer to a non-existing `streambuf` object, which might produce unexpected results. It is the responsibility of the programmer to make sure that an original `streambuf` is saved before redirection, and is restored when the redirection ends.
- Finally, at line 7, `Done` is now written to the screen again, as the redirection has been terminated.

5.8.4 Reading AND Writing streams

In order to both read and write to a stream an `fstream` object must be created. As with `ifstream` and `ofstream` objects, its 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 I use the following approach:

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

using namespace std;

int main()
{
    fstream rw("fname", ios::out | 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 will normally succeed since the file is created due to the `ios::trunc` flag. If the file already existed, the constructor will succeed. If the `ios::ate` flag would have been specified as well with `rw`'s initial construction, the first read/write action would by default have take place at `EOF`. However, `ios::ate` is not `ios::app`, so it would then still have been possible to repositioned `rw` 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::binary` 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 5.4.2.1 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::trunc);
    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 an `fstream &`, shouldn't it?

The answer is: it doesn't. The compiler casts the `fstream` object into an `ifstream` object in combination with the extraction operator, and into an `ofstream` 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 `ifstream` 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 5.4.1.2 and 5.5.1.2).

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::trunc);
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, and 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-numer 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 offet ", 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);
        fstream strings("strings", ios::trunc | ios::in | ios::out);

        cout << "enter ‘r <number>’ to read line <number> or "
                                    "w <line>’ to write a line\n"
                "or enter ‘q’ to quit.\n";

        while (true)
        {
            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()
    {                                       // r/w the file
        fstream 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[100];                   // or: char *buffer = new char[100]
        char c;
```

```
                                        // read the first ‘hello’
    cout << f.get(buffer, sizeof(buffer), 0).tellg() << endl;;
    f >> c;                             // read the ascii-z delim

                                        // and read the second ‘hello’
    cout << f.get(buffer + 6, sizeof(buffer) - 6, 0).tellg() << endl;

    buffer[5] = ' ';                    // change asciiz to ' '
    cout << buffer << endl;             // show 2 times ‘hello’
}
/*
    Generated output:
5
11
hello hello
*/
```

A completely different way to both read and write to streams can be implemented using the `streambuf` members of stream objects. All considerations mentioned so far remain valid: before a read operation following a write operation `seekg()` must be used, and before a write operation following a read operation `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 offet ", 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);
    ifstream strings_in("strings", ios::trunc | ios::in | ios::out);
    ostream  index_out(index_in.rdbuf());
```

```
    ostream  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 associate with the `streambuf` objects of existing streams are not `ifstream` or `ofstream` objects (or, for that matter, `istringstream` or `ostringstream` 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 formally introduced. Two special member functions, the constructor and the destructor, are presented.

In steps we will construct a class `Person`, which could be used in a database application to store a person's name, address and phone number.

Let's start by creating 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`. A 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
{
    std::string d_name;     // name of person
    std::string d_address;  // address field
    std::string d_phone;    // telephone number
    size_t    d_weight;   // the weight in kg.

    public:                 // interface functions
        void setName(std::string const &n);
        void setAddress(std::string const &a);
        void setPhone(std::string const &p);
        void setWeight(size_t weight);

        std::string const &name()    const;
        std::string const &address() const;
        std::string const &phone()   const;
        size_t weight()            const;
};
```

It should be noted that this terminology is frequently loosely applied. Sometimes, *class definition* is used to indicate the class interface. While the class *definition* (so, the *interface*) contains the *declarations* of its members, the actual *implementation* of these members is also referred to as the *definition* of these members. As long as the concept of the class *interface* and the class *implementation* is well distinguished, it should be clear from the context what is meant by a 'definition'.

The data fields in this class are `d_name, d_address, d_phone` and `d_weight`. All fields except `d_weight` are `string` objects. As the data fields are not given a specific *access modifier*, they are `private`, which means that they can only be accessed by the functions of the class `Person`. Alternatively, the label '`private:`' might have been used at the beginning of a private section of the class definition.

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., `name()`). Functions merely returning values stored inside the object, not allowing the caller to modify these internally stored values, are called *accessor functions*.

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.

A few remarks concerning *style*. Following *Lakos* (Lakos, J., 2001) **Large-Scale C++ Software Design** (Addison-Wesley). I suggest the following setup of class interfaces:

- All data members should have *private access rights*, and should be placed at the head of the interface.
- All data members start with `d_`, followed by a name suggesting the meaning of the variable (In chapter 10 we'll also encounter data members starting with `s_`).
- Non-private data members *do* exist, but one should be hesitant to use non-private access rights for data members (see also chapter 13).
- Two broad classes of member functions are *manipulators* and *accessor functions*. *Manipulators* allow the users of objects to actually modify the internal data of the objects. By convention, manipulators start with `set`. E.g., `setName()`.
- With *accessors*, often a `get`-prefix is encountered, e.g., `getName()`. However, following the conventions used in the **Qt** *Graphical User Interface Toolkit* (see `http://www.trolltech.com`), the `get`-prefix is *dropped*. So, rather than defining the member `getAddress()`, the function will simply be defined as `address()`.

Style conventions usually take a long time to develop. There is nothing obligatory about them, however. I suggest that readers who have compelling reasons *not* to follow the above style conventions use their own. All others should adopt the above style conventions.

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 *destructor*'s primary task is to return memory allocated by an object to the common pool when an object goes 'out of scope'. Allocation of memory is discussed in chapter 7, and destructors will therefore be discussed in depth in that chapter.

In this chapter the emphasis will be on the basic form of the `class` and on its constructors.

The constructor has by definition the same name as its 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 ensures that the constructor of a class, if defined, is called when a variable of the class, called an object, is defined ('created'). It is of course possible to define a class with no constructor at all. In that case the program will call a default constructor when a corresponding object is created. What actually happens in that case depends on the way the class has been defined. The actions of the default constructors are covered in section 6.4.1.

Objects may be defined locally or globally. However, in **C++** most objects are defined locally. Globally defined objects are hardly ever required.

When an object is defined locally (in a function), the constructor is called every time the function is called. The object's constructor is then activated at the point where the object 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();
};

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 holds true for the declaration of the constructor in the class definition, as in:

```
class Demo
{
    public:
        Demo();         // no return value here
};
```

and it 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.*

  This 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 argument, 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 `size_t d_weight` data member. These data members can be manipulated by the interface functions.

Classes (should) operate as follows:

- When the object is constructed, its data members are given 'sensible' values. Thus, objects never suffer from uninitialized values.
- The assignment to a data member (using a `set...()` function) consists of the assignment of the new value to the corresponding data member. This assignment is fully controlled by the class-designer. Consequently, the object itself is 'responsible' for its own data-integrity.
- Inspecting data members using the accessor functions simply returns the value of the requested data member. Again, this will not result in uncontrolled modifications of the object's data.

The `set...()` functions could be constructed as follows:

```
#include "person.h"                 // given earlier

// interface functions set...()
void Person::setName(string const &name)
{
    d_name = name;
```

```
}

void Person::setAddress(string const &address)
{
    d_address = address;
}

void Person::setPhone(string const &phone)
{
    d_phone = phone;
}

void Person::setWeight(size_t weight)
{
    d_weight = weight;
}
```

Next the accessor functions are defined. Note the occurence of the keyword `const` following the parameter lists of these functions: these member functions are called *const member functions*, indicating that they will not modify their *object*'s data when they're called. Furthermore, notice that the return types of the member functions returning the values of the `string` data members are `string const &` types: the `const` here indicates that the *caller* of the member function *cannot* alter the returned value itself. The caller of the accessor 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 greater detail in section 6.2. The return value of the `weight()` member function, however, is a plain `size_t`, as this can be a simple copy of the value that's stored in the `Person`'s `weight` member:

```
#include "person.h"                 // given earlier

// accessor functions ...()
string const &Person::name() const
{
    return d_name;
}

string const &Person::address() const
{
   return d_address;
}

string const &Person::phone() const
{
   return d_phone;
}

size_t Person::weight() const
{
   return d_weight;
}
```

The class definition of the `Person` class given earlier can still be used. The `set...()` and accessor functions merely implement the member functions declared in that class definition.

The following example shows the use of the class `Person`. 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 parameter is declared `const`.

Alternatively, the function `printperson()` might 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.name()     << endl <<
                "Address : " << p.address()  << endl <<
                "Phone   : " << p.phone()    << endl <<
                "Weight  : " << p.weight()   << 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 expecting three strings and an size_t may be handy: these arguments then represent, respectively, the person's name, address, phone number and weight. Such a constructor is:

```
    Person::Person(string const &name, string const &address,
```

```
                string const &phone, size_t weight)
{
    d_name = name;
    d_address = address;
    d_phone = phone;
    d_weight = weight;
}
```

The constructor must also be declared in the class interface:

```
class Person
{
    public:
        Person(std::string const &name, std::string const &address,
               std::string const &phone, size_t 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 initial values for its 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(std::string const &name, std::string const &address,
               std::string const &phone, size_t 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: as these data members themselves are objects, they are already initialized to empty strings by default. However, there is also an `size_t` data member. That member is a variable of a basic type and basic type variabes are not initialized automatically. So, unless the value of the `d_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()
{
```

```
        d_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);
        Person 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 contructed 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(std::string const &name,
                   std::string const &address = "--unknown--",
                   std::string const &phone   = "--unknown--",
                   size_t weight = 0);

            // rest of the class interface
    };
```

Often, the constructors are implemented highly similar. This results from the fact that often the constructor's parameters are defined for convenience: a constructor not requiring a phone number but requiring a weight cannot be defined using default arguments, since only the last but one parameter in the constructor defining all four parameters is not required. This cannot be solved using default argument values, but only by defining another constructor, not requiring phone to be specified.

Although some languages (e.g., **Java**) allow constructors to call constructors, this is conceptually weird. It's weird because it makes a kludge out of the constructor concept. A constructor is meant to construct an object, not to construct itself while it hasn't been constructed yet.

In **C++** the way to proceed is as follows: All constructors *must* initialize their reference data members, or the compiler will (rightfully) complain. This is one of the fundamental reasons why you can't call a constructor during a construction. Next, we have two options:

- If the body of your construction process is extensive, but (parameterizable) identical to another constructor's body, factorize! Make a private member init(maybe having params) called by the constructors. Each constructor furthermore initializes any reference data members its class may have.

- If the constructors act fundamentally differently, then there's nothing left but to construct completely different constructors.

6.1.2.1 The order of construction

The possibility to pass arguments to constructors allows us to monitor the construction of objects during a program's execution. 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>
    using namespace std;

    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 used behind the parameter list of member functions. This keyword indicates that a member function does not alter the data members of its object, but will only inspect them. These member functions are called *const member functions*. Using the example of the class `Person`, we see that the accessor functions were declared `const`:

```
class Person
{
    public:
        std::string const &name() const;
        std::string const &address() const;
        std::string const &phone() const;
};
```

This fragment illustrates that the keyword `const` appears *behind* the functions' argument lists. Note that in this situation the rule of thumb given in section 3.1.3 applies as well: 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 definitions of member functions:

```
string const &Person::name() const
{
    return d_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

```
d_name = 0;
```

in the above `const` function `name()` would result in a compilation error.

`Const` member functions exist because **C++** allows `const` objects to be created, or (used more often) 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 the definition of a `const` object of the class `Person`. 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@rug.nl", "363 9281");
```

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. As a rule of thumb it is stated here that member functions should always be given the `const` attribute, unless they actually modify the object's data.

Earlier, in section 2.5.11 the concept of function overloading was introduced. There it noted that member functions may be overloaded merely by their `const` attribute. In those cases, the compiler will use the member function matching most closely the const-qualification of the object:

- When the object is a `const` object, only `const` member functions can be used.
- When the object is not a `const` object, non-`const` member functions will be used, *unless* only a `const` member function is available. In that case, the `const` member function will be used.

An example showing the selection of (non) `const` member functions is given in the following example:

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

class X
{
    public:
        X();
        void member();
        void member() const;
};

X::X()
{}
void X::member()
{
    cout << "non const member\n";
}
void X::member() const
{
    cout << "const member\n";
}

int main()
{
    X const constObject;
    X       nonConstObject;

    constObject.member();
```

```
        nonConstObject.member();
    }
    /*
            Generated output:

        const member
        non const member
    */
```

Overloading member functions by their `const` attribute commonly occurs in the context of *operator overloading*. See chapter 9, in particular section 9.1 for details.

6.2.1 Anonymous objects

Situations exists where objects are used because they offer a certain functionality. They only exist because of the functionality they offer, and nothing in the objects themselves is ever changed. This situation resembles the well-known situation in the **C** programming language where a function pointer is passed to another function, to allow run-time configuration of the behavior of the latter function.

For example, the class `Print` may offer a facility to print a string, prefixing it with a configurable prefix, and affixing a configurable affix to it. Such a class *could* be given the following prototype:

```
    class Print
    {
        public:
            printout(std::string const &prefix, std::string const &text,
                     std::string const &affix) const;
    };
```

An interface like this would allow us to do things like:

```
    Print print;
    for (int idx = 0; idx < argc; ++idx)
        print.printout("arg: ", argv[idx], "\n");
```

This would work well, but can greatly be improved if we could pass `printout`'s invariant arguments to `Print`'s constructors: this way we would not only simplify `printout`'s prototype (only one argument would need to be passed rather than three, allowing us to make faster calls to `printout`) but we could also capture the above code in a function expecting a `Print` object:

```
    void printText(Print const &print, int argc, char *argv[])
    {
        for (int idx = 0; idx < argc; ++idx)
            print.printout(argv[idx]);
    }
```

Now we have a fairly generic piece of code, at least as far as `Print` is concerned. If we would provide `Print`'s interface with the following constructors we would be able to configure our output stream as well:

```
    Print(char const *prefix, char const *affix);
```

```
Print(ostream &out, char const *prefix, char const *affix);
```

Now `printText` could be used as follows:

```
Print p1("arg: ", "\n");             // prints to cout
Print p2(cerr, "err: --", "--\n");  // prints to cerr

printText(p1, argc, argv);          // prints to cout
printText(p2, argc, argv);          // prints to cerr
```

However, when looking closely at this example, it should be clear that both `p1` and `p2` are only used inside the `printText` function. Furthermore, as we can see from `printText`'s prototype, `printText` won't modify the internal data of the `Print` object it is using.

In situations like these it is not necessary to define objects before they are used. Instead *anonymous objects* should be used. Using anonymous objects is indicated when:

- A function parameter defines a `const` reference to an object;
- The object is *only* needed inside the function call.

Anonymous objects are defined by calling a constructor without providing a name for the constructed object. In the above example anonymous objects can be used as follows:

```
printText(Print("arg: ", "\n"), argc, argv);          // prints to cout
printText(Print(cerr, "err: --", "--\n"), argc, argv);// prints to cerr
```

In this situation the `Print` objects are constructed and immediately passed as first arguments to the `printText` functions, where they are accessible as the function's `print` parameter. While the `printText` function is executing they can be used, but once the function has completed, the `Print` objects are no longer accessible.

Anonymous objects cease to exist when the function for which they were created has terminated. In this respect they differ from ordinary local variables whose lifetimes end by the time the function block in which they were defined is closed.

6.2.1.1 Subtleties with anonymous objects

As discussed, anonymous objects can be used to initialize function parameters that are `const` references to objects. These objects are created just before such a function is called, and are destroyed once the function has terminated. This use of anonymous objects to initialize function parameters is often seen, but **C++**'s grammar allows us to use anonymous objects in other situations as well. Consider the following snippet of code:

```
int main()
{
    // initial statements
    Print("hello", "world");
    // later statements
}
```

In this example the anonymous Print object is constructed, and is immediately destroyed after its construction. So, following the 'initial statements' our Print object is constructed, then it is destroyed again, followed by the execution of the 'later statements'. This is remarkable as it shows that the standard lifetime rules do not apply to anonymous objects. Their lifetime is limited to the *statement*, rather than to the *end of the block* in which they are defined.

Of course one might wonder why a plain anonymous object could ever be considered useful. One might think of at least one situation, though. Assume we want to put *markers* in our code producing some output when the program's execution reaches a certain point. An object's constructor could be implemented so as to provide that marker-functionality, thus allowing us to put markers in our code by defining anonymous, rather than named objects.

However, **C++**'s grammar contains another remarkable characteristic. Consider the next example:

```
int main(int argc, char *argv[])
{
    Print p("", "");                        // 1
    printText(Print(p), argc, argv);    // 2
}
```

In this example a non-anonymous object p is constrcted in statement 1, which object is then used in statement 2 to *initialize* an anonymous object which, in turn, is then used to initialize printText's const reference parameter. This use of an existing object to initialize another object is common practice, and is based on the existence of a so-called *copy constructor*. A copy constructor creates an object (as it is a constructor), using an existing object's characteristics to initialize the new object's data. Copy constructors are discussed in depth in chapter 7, but presently merely the concept of a copy constructor is used.

In the last example a copy constructor was used to initialize an anonymous object, which was then used to initialize a parameter of a function. However, when we try to apply the same trick (i.e., using an existing object to initialize an anonymous object) to a plain statement, the compiler generates an error: the object p can't be redefined (in statement 3, below):

```
int main(int argc, char *argv[])
{
    Print p("", "");                        // 1
    printText(Print(p), argc, argv);    // 2
    Print(p);                               // 3 error!
}
```

So, using an existing object to initialize an anonymous object that is used as function argument is ok, but an existing object can't be used to initialize an anonymous object in a plain statement?

The answer to this apparent contradiction is actually found in the compiler's error message itself. At statement 3 the compiler states something like:

```
error: redeclaration of 'Print p'
```

which solves the problem, by realizing that within a compound statement objects and variables may be defined as well. Inside a compound statement, a *type name* followed by a variable name is the grammatical form of a variable definition. *Parentheses* can be used to break priorities, but if there are no priorities to break, they have no effect, and are simply ignored by the compiler. In statement 3 the parentheses allowed us to get rid of the blank that's required between a type name and the variable name, but to the compiler we wrote

```
        Print (p);
```

which is, since the parentheses are superfluous, equal to

```
        Print p;
```

thus producing `p`’s redeclaration.

As a further example: when we define a variable using a basic type (e.g., `double`) using superfluous parentheses the compiler will quietly remove these parentheses for us:

```
    double ((((a))));       // weird, but ok.
```

To summarize our findings about anonymous variables:

- Anonymous objects are great for initializing `const` reference parameters.
- The same syntaxis, however, can also be used in stand-alone statements, in which they are interpreted as variable definitions if our intention actually was to initialize an anonymous object using an existing object.
- Since this may cause confusion, it’s probably best to restrict the use of anonymous objects to the first (and main) form: initializing function parameters.

6.3 The keyword ‘inline’

Let us take another look at the implementation of the function `Person::name()`:

```
    std::string const &Person::name() const
    {
        return d_name;
    }
```

This function is used to retrieve the name field of an object of the class `Person`. In a code fragment like:

```
    Person frank("Frank", "Oostumerweg 17", "403 2223");

    cout << frank.name();
```

the following actions take place:

- The function `Person::name()` 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, without ever actually calling a function `name()`. This can be realized using `inline` functions.

6.3.1 Defining members inline

Inline functions may be implemented *in the class interface itself*. For the class `Person` this results in the following implementation of `name()`:

```
class Person
{
    public:
        std::string const &name() const
        {
            return d_name;
        }
};
```

Note that the inline code of the function `name()` now literally occurs inline in the interface of the class `Person`. The keyword `const` occurs after the function declaration, and before the code block.

Although members can be defined inside the class interface itself, it should be considered bad practice because of the following considerations:

- Defining functions inside the interface confuses the interface with the implementation. The interface should merely document what functionality the class offers. Mixing member declarations with implementation detail complicates understanding the interface. Readers will have to skip over implementation details which takes time and makes it hard to grab the 'broad picture', and thus to understand at a glance what functionality the class's objects are offering.
- Although members that are eligible for inline-coding should remain inline, situations do exist where members migrate from an inline to a non-inline definition. The in-class inline definition still needs editiing (sometimes considerable editing) before a non-inline definition is ready to be compiled. This additional editing is undesirable.

Because of the above considerations inline members should not be defined within the class interface. Rather, they should be defined *below* the class interface. The `name()` member of the `Person` class is therefore preferably defined as follows:

```
class Person
{
    public:
        std::string const &name() const;
};

inline std::string const &Person::name() const
{
    return d_name;
}
```

This version of the `Person` class clearly shows that:

- the class interface itself only contains a declaration
- the inline implementation can easily be redefined as a non-inline implementation by removing the `inline` keyword and including the appropriate class-header file. E.g.,

```
    #include "person.h"
```

```
std::string const &Person::name() const
{
    return d_name;
}
```

Defining members inline has the following effect: Whenever an inline function is called in a program statement, the compiler may *insert the function's body* at the location of the function call. The function itself may never actually be called. Consequently, the function call is prevented, but the function's body appears as often in the final program as the inline function is actually called.

This construction, where the function code itself is inserted rather than a call to the function, is called an inline function. Note that using inline functions may result in multiple occurrences of the code of those functions in a program: one copy for each invocation of the inline function. This is probably ok if the function is a small one, and needs to be executed fast. It's not so desirable if the code of the function is extensive. The compiler knows this too, and 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. As a rule of thumb: members should only be defined inline if they are small (containing a single, small statement) and if it is highly unlikely that their definition will ever change.

6.3.2 When to use inline functions

When should `inline` functions be used, and when not? There are some rules of thumb which may be followed:

- In general `inline` functions should **not** be used. *Voilà*; that's simple, isn't it?
- 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 to perform for such optimizations.
- `inline` functions can be used when member functions consist of one very simple statement (such as the return statement in the function `Person::name()`).
- 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 spent during a function call is long compared to the code in the function. An example of an `inline` function which will hardly have any effect on the program's speed is:

  ```
  void Person::printname() const
  {
      cout << d_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 an 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 the keyword inline is not really a *command* to the compiler. Rather, it is a *request* the compiler may or may not grant.

6.4 Objects inside objects: composition

Often objects are used as data members in class definitions. This is called *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** a struct or union field is commonly used in other compound types.

The initialization of composed objects deserves some special 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 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 *name, char const *address,
                char const *phone, size_t weight)
:
    d_name(name),
    d_address(address),
    d_phone(phone),
    d_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* initialized automatically.

Member initialization can, however, also be used for primitive data members, like `int` and `double`. The above example shows the initialization of the data member `d_weight` from the parameter `weight`. Note that with member initializers the data member could even have the same name as the constructor parameter (although this is deprecated): with member initialization there is no ambiguity and the first (left) identifier in, e.g., `weight(weight)` is 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 often 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 efficent.

```
Person::Person(char const *name, char const *address,
                char const *phone, size_t weight)
{
    d_name = name;
    d_address = address;
    d_phone = phone;
    d_weight = weight;
}
```

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 *name, char const *address,
                char const *phone, char const *birthday,
                size_t weight)
:
    d_name(name),
    d_address(address),
    d_phone(phone),
    d_birthday(birthday),       // assume: string const d_birthday
    d_weight(weight)
{}
```

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. Bluntly passing an object (i.e., by value) might not be a very good idea, since the object must be copied into the `Configfile` parameter, and then a data member of the `Process` class can be used to make the `Configfile` object accessible throughout the `Process` class. This might involve yet another object-copying task, as in the following situation:

  ```
  Process::Process(Configfile conf)   // a copy from the caller
  {
      d_conf = conf;                  // copying to conf_member
  }
  ```

- The copy-instructions can be avoided if *pointers* to the `Configfile` objects are used, as in:

  ```
  Process::Process(Configfile *conf)  // pointer to external object
  {
      d_conf = conf;                  // d_conf is a Configfile *
  }
  ```

  This construction as such is ok, but forces us to use the '`->`' field selector operator, rather than the '`.`' operator, which is (disputably) awkward: conceptually one tends to think of the `Configfile` object as an object, and not as a pointer to an object. In **C** this would probably have been the preferred method, but in **C++** we can do better.

- Rather than using value or pointer parameters, the `Configfile` parameter could be defined as a *reference parameter* 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 initialization of the `Configfile &d_conf` reference data member:

```
Process::Process(Configfile &conf)
{
    d_conf = conf;        // wrong: no assignment
}
```

The statement `d_conf = 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 (`d_conf`). 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 `d_conf` refer? To no variable, since we haven’t initialized `d_conf`. After all, the whole purpose of the statement `d_conf = conf` was to initialize `d_conf`....

So, how do we proceed when `d_conf` must be initialized? In this situation we once again use the member initializer syntax. The following example shows the correct way to initialize `d_conf`:

```
    Process::Process(Configfile &conf)
    :
        d_conf(conf)      // initializing reference member
    {}
```

Note that this syntax must be used in all cases where reference data members are used. If `d_ir` would be an `int` reference data member, a construction like

```
    Process::Process(int &ir)
    :
        d_ir(ir)
    {}
```

would have been called for.

6.5 The keyword ‘mutable’

Earlier, in section 6.2, the concepts of `const` member functions and `const` objects were introduced.

C++, however, allows the construction of objects which are, in a sense, neither `const` objects, nor *non*-`const` objects. Data members which are defined using the keyword `mutable`, can be modified by `const` member functions.

An example of a situation where `mutable` might come in handy is where a `const` object needs to register the number of times it was used. The following example illustrates this situation:

```
#include <string>
#include <iostream>
#include <memory>


class Mutable
{
    std::string d_name;
    mutable int d_count;                // uses mutable keyword

    public:
        Mutable(std::string const &name)
        :
            d_name(name),
            d_count(0)
```

```
        {}

        void called() const
        {
            std::cout << "Calling " << d_name <<
                                    " (attempt " << ++d_count << ")\n";
        }
};


int main()
{
    Mutable const x("Constant mutable object");

    for (int idx = 0; idx < 4; idx++)
        x.called();                     // modify data of const object
}

/*
    Generated output:

    Calling Constant mutable object (attempt 1)
    Calling Constant mutable object (attempt 2)
    Calling Constant mutable object (attempt 3)
    Calling Constant mutable object (attempt 4)
*/
```

The keyword `mutable` may also be useful in classes implementing, e.g., reference counting. Consider a class implementing reference counting for textstrings. The object doing the reference counting might be a `const` object, but the class may define a copy constructor. Since `const` objects can't be modified, how would the copy constructor be able to increment the reference count? Here the `mutable` keyword may profitably be used, as it can be incremented and decremented, even though its object is a `const` object.

The advantage of having a `mutable` keyword is that, in the end, the programmer decides which data members can be modified and which data members can't. But that might as well be a disadvantage: having the keyword `mutable` around prevents us from making rigid assumptions about the stability of `const` objects. Depending on the context, that may or may not be a problem. In practice, `mutable` tends to be useful only for internal bookkeeping purposes: accessors returning values of mutable data members might return puzzling results to clients using these accessors with `const` objects. In those situations, the nature of the returned value should clearly be documented. As a rule of thumb: do not use `mutable` unless there is a very clear reason to divert from this rule.

6.6 Header file organization

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 frequently has 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 the source file of the function `File::gets()` proceeds as follows:

- The header file `project/file.h` is opened to be read;
- `_File_h_` is defined
- The header file `project/string.h` is opened to be read
- `_String_h_` is defined
- The header file `project/file.h` is (again) opened to be read
- Apparently, `_File_h_` is already defined, so the remainder of `project/file.h` is skipped.
- The interface of the class `String` is now parsed.
- In the class interface a reference to a `File` object is encountered.
- As the `class File` hasn't been parsed yet, a `File` is still an undefined type, and the compiler quits with an error.

The solution for this problem is to use a *forward class reference before* the class interface, and to include the corresponding class header file *after* the class interface. 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 classes are involved and with (non-inline) member functions having class-type return values or parameters.

Note that this setup doesn't work with composition, nor with inline member functions. Assume the class `File` has a *composed* data member of the class `String`. In that case, the class interface of the class `File` *must* include the header file of the class `String` before the class interface itself, because otherwise the compiler can't tell how big a `File` object will be, as it doesn't know the size of a `String` object once the interface 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 interface 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
{
    String d_line;              // composition !

    public:
        void gets(String &string);
};
#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 remaining header files (appearing below the class interface itself) are required only because they are used by the class's source files.

This approach allows us to introduce yet another refinement:

- Header files defining a class interface should *declare* what can be declared before defining the class interface itself. So, classes that are mentioned in a class interface should be specified using forward declarations *unless*

- They are a *base class* of the current class (see chapter 13);
- They are the class types of composed data members;
- They are used in inline member functions.

In particular: additional actual header files are *not* required for:

- class-type return values of functions;
- class-type value parameters of functions.

Header files of classes of objects that are either composed or inherited or that are used in inline functions, *must* be known to the compiler before the interface of the current class starts. The information in the header file itself is protected by the `#ifndef ... #endif` construction introduced in section 2.5.9.

- Program sources in which the class is used only need to include this header file. *Lakos*, (2001) refines this process even further. See his book **Large-Scale C++ Software Design** for further details. This header file should be made available in a well-known location, such as a directory or subdirectory of the standard `INCLUDE` path.

- For the implementation of the member functions the class's header file is required and usually other header files (like `#include <string>`) as well. The class header file itself as well as these additional header files should be included in a separate internal header file (for which the extension `.ih` ('internal header') is suggested).

  The `.ih` file should be defined in the same directory as the source files of the class, 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`:

```
#ifndef _File_h_
#define _File_h_

#include <fstream>      // for composed 'ifstream'

class Buffer;           // forward reference

class File              // class interface
{
    ifstream d_instream;

    public:
        void gets(Buffer &buffer);
};
#endif
```

  - Second part, e.g., ~/myproject/file/file.ih, where all sources of the class File are stored:

```
#include <myheaders/file.h> // make the class File known
```

```
#include <buffer.h>         // make Buffer known to File
#include <string>           // used by members of the class
#include <sys/stat.h>       // File.
```

6.6.1 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 `Inserter cout` is declared, then `special::cout` is of course a different object than `std::cout`. Now, if a class `Flaw` is constructed, in which the constructor expects a reference to a `special::Inserter`, then the class should be constructed as follows:

```
class special::Inserter;

class Flaw
{
public:
    Flaw(special::Inserter &ins);
};
```

Now the person designing the class `Flaw` may be in a lazy mood, and might get bored by continuously having to prefix `special::` before every entity from that namespace. So, the following construction is used:

```
using namespace special;

class Inserter;

class Flaw
{
public:
    Flaw(Inserter &ins);
};
```

This works fine, up to the point where somebody wants to include `flaw.h` in other source files: because of the `using` directive, this latter person is now by implication also `using namespace special`, which could produce unwanted or unexpected effects:

```
#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 intented 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 `int`s 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 explicitly checking for 0 return values.

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'

C++ defines two operators to allocate and deallocate 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 pointervalue.

The operator `new` can be used to allocate primitive types and to allocate objects. When a non-class type is allocated (a primitive type or a `struct` type without a constructor), the allocated memory is *not* guaranteed to be initialized to 0. Alternatively, an initialization expression may be provided:

```
int *v1 = new int;          // not guaranteed to be initialized to 0
int *v1 = new int();        // initialized to 0
int *v2 = new int(3);       // initialized to 3
int *v3 = new int(3 * *v2); // initialized to 9
```

When class-type 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 9.9 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]`. With non-class types (primitive types, `struct` types without constructors, pointer types) the returned allocated block of memory is *not* guaranteed to be initialized to 0.

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 9.9.

Similar to **C**, and without resorting to the operator `new[]`, arrays of variable size can also be constructed as *local 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, comparable 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 `Objects` is:

```
Object *op = new Object[10];
delete[] op;
```

Realize that `delete[]` only has an additional effect if the block of memory to be deallocated consists of *objects*. With pointers or values of primitive types 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 9.9 redefining `delete[]` is discussed. The rule of thumb is: if `new[]` was used, also use `delete[]`.

7.1.3 Enlarging arrays

Once allocated, all arrays are arrays of fixed size. 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>
using namespace std;

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;                       // 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 existing 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 has no return value.

The destructor for the class `Person` is thus 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 other members are declared.

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
{
    char *d_name;
    char *d_address;
    char *d_phone;

    public:
```

```
            Person();
            Person(char const *name, char const *address,
                   char const *phone);
            ~Person();

            char const *name() const;
            char const *address() const;
            char const *phone() const;
    };

        inline Person::Person()
        {}

/*
    person.ih contains:

    #include "person.h"
    char const *strdupnew(char const *org);
*/
```

The task of the constructor is to initialize the data fields of the object. E.g, the constructor is defined as follows:

```
    #include "person.ih"

    Person::Person(char const *name, char const *address, char const *phone)
    :
        d_name(strdupnew(name)),
        d_address(strdupnew(address)),
        d_phone(strdupnew(phone))
    {}
```

In this class the destructor is necessary to prevent that memory, allocated for the fields `d_name`, `d_address` and `d_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 a 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 to which its three data members point. The implementation of the destructor would therefore be:

```
    #include "person.ih"

    Person::~Person()
    {
        delete d_name;
        delete d_address;
        delete d_phone;
    }
```

In the following example a Person object is created, and its data fields are printed. After this the showPerson() function stops, resulting in 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");
    Person *frank = new Person("Frank", "Oostumerweg", "050 403 2223");

    cout << karel.name()     << ", " <<
            karel.address()  << ", " <<
            karel.phone()    << endl <<
            frank->name()    << ", " <<
            frank->address() << ", " <<
            frank->phone()   << 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, one of the advantages of the operators new and delete over functions like malloc() and free() is that new and delete call the corresponding constructors and destructors. 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, initializing 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 be used here 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"
    using namespace std;

    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 destructors of the individual `Person` objects are called if the `delete[]` syntax is followed, while only the first object's destructor is called 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"
using namespace std;

Person::~Person()
{
    cout << "Person destructor called" << endl;
}

int main()
{
    Person **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, as the `[]` is 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 = 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;
    }
    /*
        Generated 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>
    using namespace std;

    class Informer
    {
        public:
            ~Informer();
    };

        inline Informer::~Informer()
        {
            cout << "destructor called\n";
        }

    class Wrapper
    {
        Informer *d_i;

        public:
            Wrapper();
            ~Wrapper();
    };

        inline Wrapper::Wrapper()
        :
            d_i(new Informer())
        {}
        inline Wrapper::~Wrapper()
        {
            delete d_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 throws a *(bad_alloc) exception* () (see section 8.10), terminating the program. Consequently, in the default case it is never necessary to check the return value of the operator `new`. This default behavior may be modified in various ways. One way to modify this default behavior is to redefine the function handling failing memory allocation. However, any user-defined function must comply with the following prerequisites:

- it has no arguments, and
- it returns no value

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()`.

The implementation of an error function is illustrated below[1]:

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

void outOfMemory()
{
    cout << "Memory exhausted. Program terminates." << endl;
    exit(1);
}

int main()
{
    long allocated = 0;

    set_new_handler(outOfMemory);       // install error function

    while (true)                        // eat up all memory
    {
        new int [100000];
        allocated += 100000 * sizeof(int);
        cout << "Allocated " << allocated << " bytes\n";
    }
}
```

[1] This implementation applies to the Gnu **C/C++** requirements. The actual try-out of the program given in the example is not encouraged, as it will slow down the computer enormously due to the resulting use of the operating system's *swap area*.

After installing the error function it is automatically invoked when memory allocation fails, and the program exits. Note that memory allocation may fail in indirectly called code as well, e.g., when constructing or using streams or when strings are duplicated by low-level functions.

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. An example using `new` to duplicate a string, was given in a rewrite of the function `strdup()` (see section 7).

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 for non-class type data members is a straight byte-by-byte copy from one data member 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.name()       << endl <<
            "Address:  " << tmp.address()    << endl <<
            "Phone:    " << tmp.phone()      << 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 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 contains 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 effec-

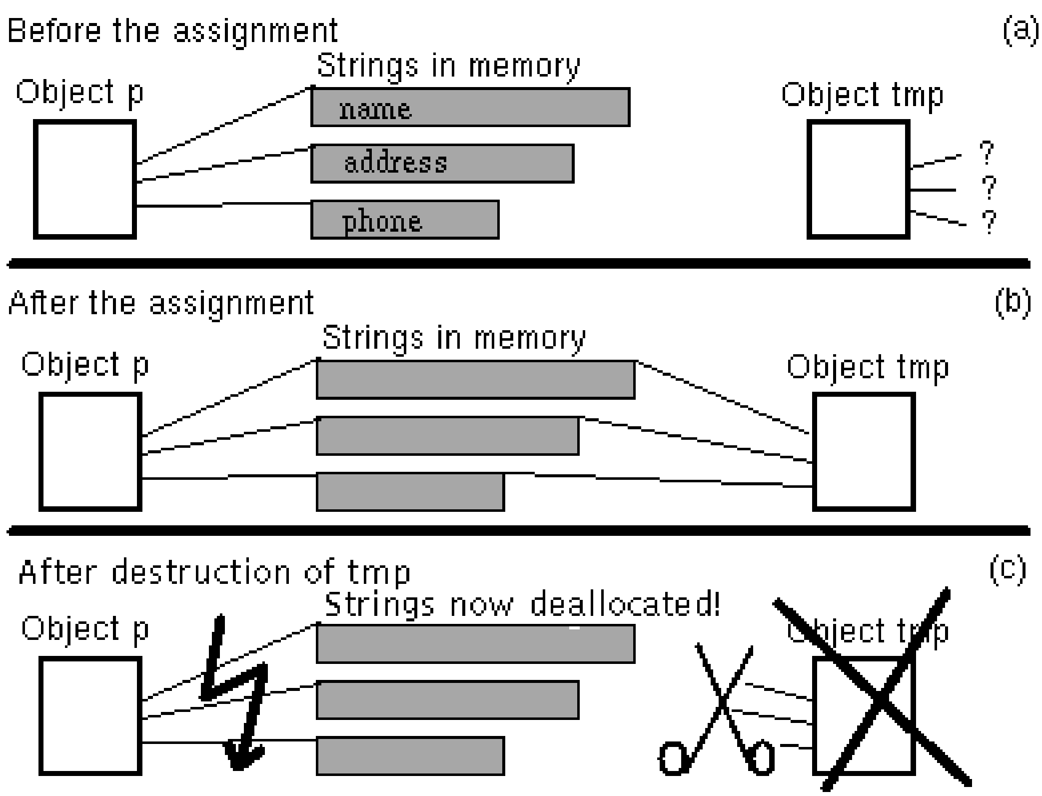

Figure 7.1: Private data and public interface functions of the class Person, using byte-by-byte assignment

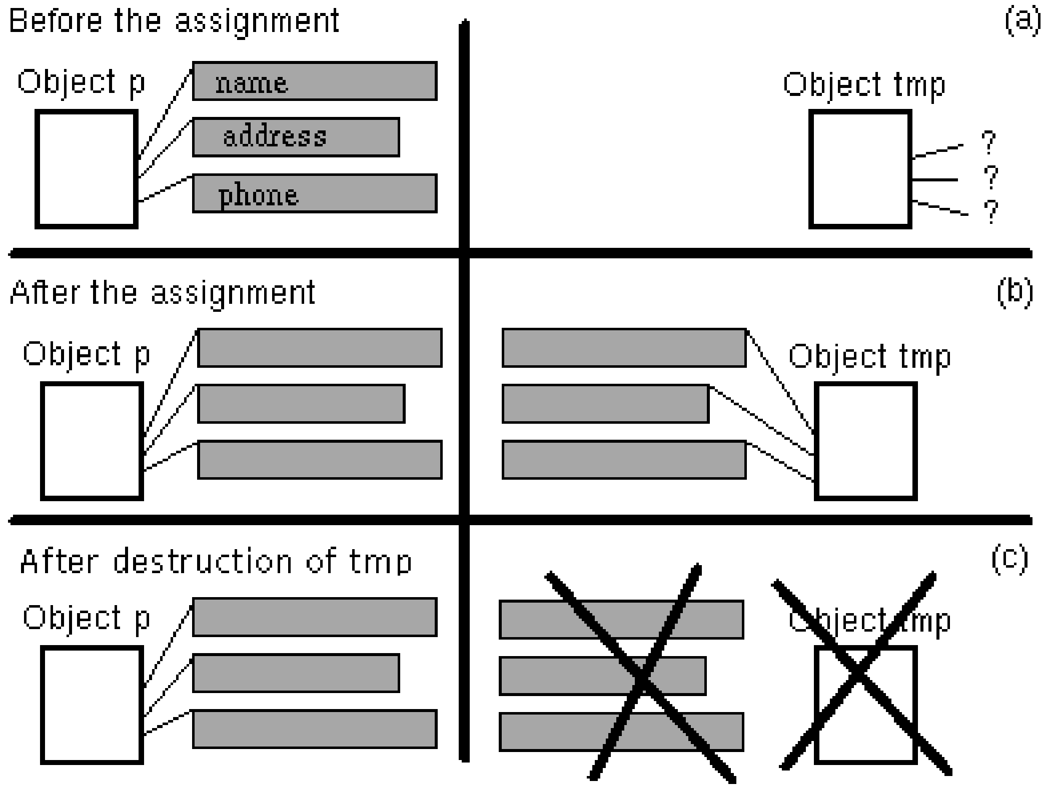

Figure 7.2: Private data and public interface functions of the class Person, using the 'correct' assignment.

tively become *wild pointers*, as they don't point to allocated memory anymore. In general it can be concluded that

> *every class containing pointer data members is a potential candidate for trouble.*

Fortunately, it is possible to prevent these troubles, as discussed 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 bytewise. 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 member function to handle assignments of objects of the class `Person`. The purpose of this member 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 d_name;
```

```
        delete d_address;
        delete d_phone;

        // now copy the other Person's data
        d_name = strdupnew(other.d_name);
        d_address = strdupnew(other.d_address);
        d_phone = strdupnew(other.d_phone);
    }
```

Using 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.name()       << endl <<
                "Address:  " << tmp.address()    << endl <<
                "Phone:    " << tmp.phone()      << endl;

        // now it doesn't matter that tmp gets destroyed..
    }
```

By 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 to be used with streams (like `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, I consider overloading the insertion (<<) and extraction (>>) operators in the context of streams ill-chosen: the stream operations do not have anything in common with the bitwise shift operations.

7.3.1.1 The member 'operator=()'

To achieve operator overloading in the context of a class, the class is simply expanded with a (usually *public*) member function naming the particular operator. That member function is thereupon defined.

For example, to overload the assignment operator =, a function `operator=()` must be defined. Note that the function name consists of two parts: the keyword `operator`, followed by the operator itself. When we augment a class interface with a member function `operator=()`, then that operator is *redefined* for the class, which prevents the default operator from being used. Previously (in section 7.3.1) the function `assign()` was offered to solve the memory-problems resulting from using the default assignment operator. However, instead of using an ordinary member function it is much more common in **C++** to define a dedicated *operator* for these special cases. So, the earlier `assign()` member may be redefined as follows (note that the member `operator=()` presented below is a first, rather unsophisticated, version of the overloaded assignment operator. It will be improved shortly):

```
class Person
{
    public:                             // extension of the class Person
                                        // earlier members are assumed.
        void operator=(Person const &other);
};
```

and its implementation could be

```
void Person::operator=(Person const &other)
{
    delete d_name;                      // delete old data
    delete d_address;
    delete d_phone;

    d_name = strdupnew(other.d_name);   // duplicate other's data
    d_address = strdupnew(other.d_address);
    d_phone = strdupnew(other.d_phone);
}
```

The actions of this member function are similar to those of the previously proposed function `assign()`, but now its *name* ensures 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");
Person copy;

copy = pers;                // first possibility
copy.operator=(pers);       // second possibility
```

Actually, the second possibility, explicitly calling `operator=()`, is not used very often. However, the code fragment *does* illustrate two ways to call the same overloaded operator member 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[2].

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 *const this;
```

A member function like `name()`, which returns the `name` field of a `Person`, could therefore be implemented in two ways: with or without the `this` pointer:

```
char const *Person::name()   // implicit usage of ‘this’
{
    return d_name;
}

char const *Person::name()   // explicit usage of ‘this’
{
    return this->d_name;
}
```

The `this` pointer is not frequently used explicitly. However, situations do exist where the `this` pointer is actually required (cf. chapter 15).

7.4.1 Preventing self-destruction using ‘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 one;
```

[2] Note that ‘`this`’ is not available in the not yet discussed `static` member functions.

```
Person two;
Person *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 d_name;
        delete d_address;
        delete d_phone;

        d_name = strdupnew(other.d_name);
        d_address = strdupnew(other.d_address);
        d_phone = strdupnew(other.d_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 assignment operator associates from right to left. I.e., in statements like:

```
a = b = c;
```

the expression `b = c` is evaluated first, and the result is assigned to `a`.

So far, the implementation of the overloaded assignment operator 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 concatenated assignments are still not allowed.

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 in 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 d_address;
        delete d_name;
        delete d_phone;

        d_address = strdupnew(other.d_address);
        d_name = strdupnew(other.d_name);
        d_phone = strdupnew(other.d_phone);
    }
    // return current object. 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)
Person karel2;                                             // see (2)
Person karel3 = karel;                                     // see (3)

int main()
{
    karel2 = karel3;                                    // see (4)
    return 0;
}
```

- Statement 1: this shows an initialization. The object `karel` is initialized with appropriate texts. This construction of `karel` therefore uses the constructor expecting three `char const *` arguments.

  Assume a `Person` constructor is available having only one `char const *` parameter, e.g.,

  ```
  Person::Person(char const *n);
  ```

  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*.

- 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 actual constructor that is to be used can be deduced from the constructor's argument list. The assignment operator may be used if the constructor has only one parameter (and also when remaining parameters have default argument values).

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)
{
    d_name    = strdupnew(other.d_name);
    d_address = strdupnew(other.d_address);
    d_phone   = strdupnew(other.d_phone);
}
```

The actions of copy constructors are comparable to those of the overloaded assignment operators: an object is *duplicated*, so that it will contain 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.

Apart from 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 initialized using 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.name() << 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 person()
  ```

```
    {
        string name;
        string address;
        string 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 `person()` terminates.

To demonstrate that copy constructors are not called in all situations, consider the following. We could rewrite the above function `person()` to the following form:

```
Person person()
{
    string name;
    string address;
    string 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 person()`).

As an other example, once again assuming the availability of a `Person(char const *name)` constructor, consider:

```
Person namedPerson()
{
    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 if not, then the default copy constructor will still be available when another constructor is defined.

7.5.1 Similarities between the copy constructor and operator=()

The similarities between the copy constructor and 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 implemented 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
{
    char *d_name;
    char *d_address;
    char *d_phone;

    public:
        Person(Person const &other);
        ~Person();
        Person &operator=(Person const &other);
    private:
        void copy(Person const &other);     // new members
        void destroy(void);

};
```

- Next, the functions `copy()` and `destroy()` are constructed:

```
void Person::copy(Person const &other)
{
    d_name = strdupnew(other.d_name);       // unconditional copying
    d_address = strdupnew(other.d_address);
    d_phone = strdupnew(other.d_phone);
}

void Person::destroy()
{
    delete d_name;                        // unconditional deletion
    delete d_address;
    delete d_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 by the copy constructor's body *except* for the initialization of 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 either, as reference members cannot be given another value once initialized. An object having reference data members is inseparately attached to its referenced object(s) once it has been constructed.

7.5.2 Preventing certain members from being used

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 do 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 stil 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, in particular with the overloaded assignment operator, 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 members, addressing allocated memory, are potential sources of memory leaks. The two extensions introduced in this chapter represent the standard way to prevent these memory leaks.

The simple conclusion is therefore: classes whose objects allocate memory which is used by these objects themselves, should implement a *destructor*, an *overloaded assignment operator* and a *copy constructor* as well.

Chapter 8

Exceptions

C supports several ways in which a program can 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, of the mentioned alternatives, the `setjmp()` and `longjmp()` approach isn't frequently seen in **C++** (or even in **C**) programs, due to the fact that the program flow is completely disrupted.

C++ offers *exceptions* as the preferred alternative to `setjmp()` and `longjmp()` are. Exceptions allow **C++** programs to perform a controlled non-local return, 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 is not disastrous enough for a program to terminate completely. Also, exceptions provide a flexible layer of control between the short-range `return` and the crude `exit()`.

In this chapter exceptions and their syntax will be introduced. First an example of the different impacts exceptions and `setjmp()` and `longjmp()` have on a program will be given. Then the discussion will dig into the formalities exceptions.

8.1 Using exceptions: syntax elements

With exceptions the following syntactical elements are used:

- `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 `main()`, and its member `Outer::fun()` is called. Then, in `Outer::fun()` an `Inner` object is constructed. Having constructing the `Inner` object, its member `Inner::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>
using namespace std;

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";
}

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 entirely as expected, and it shows 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 Anachronisms: '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 <cstdlib>

using namespace std;

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
    */
```

The output produced by this program clearly shows 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.

As this example shows that the destructors of objects can easily be skipped when `longjmp()` and `setjmp()` are used, these function should be *avoided* 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>
using namespace std;

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 function `Inner::fun()` terminates, albeit with an exception rather than by a call to `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 this has raised your appetite for exceptions, since it was shown 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, resulting 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 thrown by these functions leave 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>
using namespace std;

class Object
{
    string d_name;

    public:
        Object(string name)
        :
            d_name(name)
        {
            cout << "Object constructor of " << d_name << "\n";
        }
        Object(Object const &other)
        :
            d_name(other.d_name + " (copy)")
        {
            cout << "Copy constructor for " << d_name << "\n";
        }
        ~Object()
        {
            cout << "Object destructor of " << d_name << "\n";
        }
        void fun()
        {
            Object toThrow("'local object'");

            cout << "Object fun() of " << d_name << "\n";
            throw toThrow;
```

```
            }
            void hello()
            {
                cout << "Hello by " << d_name << "\n";
            }
    };

    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 several 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. Threfore, 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 repeatedly calling the copy constructor 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 occur in which it is required to inspect a thrown exception. Then, 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 into 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 is processed there. If it is not possible to process the exception at the middle level, it is passed on, unaltered, to a more superficial level, where the really tough exceptions are handled.

By placing an *empty* `throw` statement in the 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))
        throw;

    handleTheMessage(exception);
}
```

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 any of 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
    )
        throw;                      // Pass on other types of Exceptions

    e->process();               // Process a message or a warning
    delete e;
}
```

Due to polymorphism (see chapter 14), `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 directly in the `try`-block. It may, for example, be placed in a function, called from within the `try`-block.

The keyword `try` is followed by a set of curly braces, acting like a standard **C++** compound statement: multiple statements and definitions may be placed here.

It is possible (and very common) to create *levels* in which exceptions may be thrown. For example, `main()`'s code is surrounded by a `try`-block, forming an outer level in which exceptions can be handled. Within `main()`'s `try`-block, functions are called which may also contain `try`-blocks, forming the next level in which exceptions may be generated. 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 consisting of but one parameter, which is the type of the exception 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. 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.

Close attention should be paid to the nature of the parameter of the exception handler, to make sure that 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, `int`s, 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.

To some extent the *order* of the exception handlers is important. When an exception is thrown, the first exception handler matching the type of the thrown exception is used and remaining exception handlers are ignored. So only one exception handler following a `try`-block will be executed. Normally this is no problem: the thrown exception is of a certain type, and the correspondingly typed catch-handler will catch it. For example, if exception handlers are defined for `char *`s and `void *`s then ASCII-Z strings will be caught by the latter handler. Note that a `char *` can also be considered a `void *`, but even so, an ASCII-Z string will be handled by a `char *` handler, and not by a `void *` handler. This is true in general: handlers should be designed very type specific to catch the correspondingly typed exception. For example, `int`-exceptions are not caught by `double`-catchers, `char`-exceptions are not caught by `int`-catchers. Here is a little example illustrating that the order of the catchers is not important for types not having any hierarchical relation to each other (i.e., `int` is not derived from `double`; `string` is not derived from ASCII-Z):

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

int main()
{
    while (true)
    {
        try
        {
            string s;
            cout << "Enter a,c,i,s for ascii-z, char, int, string "
                                                      "exception\n";
            getline(cin, s);
            switch (s[0])
            {
                case 'a':
                    throw "ascii-z";
                case 'c':
                    throw 'c';
                case 'i':
                    throw 12;
                case 's':
                    throw string();
            }
        }
        catch (string const &)
        {
            cout << "string caught\n";
        }
        catch (char const *)
        {
            cout << "ASCII-Z string caught\n";
        }
        catch (double)
        {
            cout << "isn't caught at all\n";
```

```
        }
        catch (int)
        {
            cout << "int caught\n";
        }
        catch (char)
        {
            cout << "char caught\n";
        }
    }
}
```

As an alternative to constructing different types of exception handlers for different types of exceptions, a specific class can be designed whose objects contain information about 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 `delete e` statement in the above code indicates that the `Exception` object was created dynamically.

When the code of an exception handler has been processed, execution continues beyond the last exception handler directly following that `try`-block (assuming the handler doesn't itself use flow control statements (like `return` or `throw`) to break the default flow of execution). From this, 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 from the `try`-block and an appropriate exception handler is available, then 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.

All statements in a `try` block appearing below an executed `throw`-statement will be ignored. 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 an exception may be realized using various degrees of sophistication. For example, it's possible to use the operator `new`; 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 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 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>
using namespace std;

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 defined elsewhere may be linked to code using 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. Declarations of such functions may contain a *function throw list* or *exception specification list*, in which the types of the exceptions that can be thrown by the function are specified. For example, a function that could throw '`char *`' and '`int`' exceptions can be declared as

```
void exceptionThrower() throw(char *, int);
```

If specified, a function throw list appears immediately beyond the function header (and also beyond a possible `const` specifier), and, noting that throw lists 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 all cases, the function header used in the function definition must exactly match the function header that is used in the declaration, e.g., including a possible empty function throw list.

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 those mentioned in the function throw list.

For example, consider the declarations and definitions in the following program:

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

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 program's designer.

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:

- `iostate exceptions()`: this member returns the state flags for which the stream will throw exceptions,
- `void exceptions(iostate state)`: this member will throw an exception when state `state` is observed.

In the context of the I/O library, exceptions are objects 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 exceptional situations. Therefore, we think it is questionable to have stream objects throw exceptions for rather standard situations like `EOF`. Using exceptions to handle input errors 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 would be a more appropriate action. Here is an example showing the use of exceptions in an interactive program, expecting numbers:

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

int main()
{
    cin.exceptions(ios::failbit);

    while (true)
    {
        try
        {
            cout << "enter a number: ";

            int value;

            cin >> value;
            cout << "you entered " << value << endl;
        }
        catch (ios::failure const &problem)
        {
            cout << problem.what() << endl;
            cin.clear();
            string s;
            getline(cin, s);
        }
    }
}
```

8.8 Exceptions in constructors and destructors

Only constructed objects are eventually destroyed. Although this may sound like a truism, there is a subtlety here. If the construction of an object fails for some reason, the object's destructor will not be called once the object goes out of scope. This could happen if an *uncaught exception* is generated by the constructor. If the exception is thrown *after* the object has allocated some memory, then its destructor (as it isn't called) won't be able to delete the allocated block of memory. A *memory leak* will be the result.

The following example illustrates this situation in its prototypical form. The constructor of the class `Incomplete` first displays a message and then throws an exception. Its destructor also displays a message:

```
class Incomplete
```

```
{
    public:
        Incomplete()
        {
            cerr << "Allocated some memory\n";
            throw 0;
        }
        ~Incomplete()
        {
            cerr << "Destroying the allocated memory\n";
        }
};
```

Next, `main()` creates an `Incomplete` object inside a `try` block. Any exception that may be generated is subsequently caught:

```
int main()
{
    try
    {
        cerr << "Creating ‘Incomplete’ object\n";
        Incomplete();
        cerr << "Object constructed\n";
    }
    catch(...)
    {
        cerr << "Caught exception\n";
    }
}
```

When this program is run, it produces the following output:

```
Creating ‘Incomplete’ object
Allocated some memory
Caught exception
```

Thus, if `Incomplete`'s constructor would actually have allocated some memory, the program would suffer from a memory leak. To prevent this from happening, the following countermeasures are available:

- Exceptions should not leave the constructor. If part of the constructor's code may generate exceptions, then this part should itself be surrounded by a `try` block, catching the exception within the constructor. There may be good reasons for throwing exceptions out of the constructor, as that is a direct way to inform the code using the constructor that the object has not become available. But before the exception leaves the constructor, it should be given a chance to delete memory it already has allocated. The following skeleton setup of a constructor shows how this can be realized. Note how any exception that may have been generated is rethrown, allowing external code to inspect this exception too:

  ```
  Incomplete::Incomplete()
  {
      try
      {
  ```

```
            d_memory = new Type;
            code_maybe_throwing_exceptions();
        }
        catch (...)
        {
            delete d_memory;
            throw;
        }
    };
```

- Exceptions might be generated while initializing members. In those cases, a `try` block within the constructor's body has no chance to catch such exceptions. When a class uses pointer data members, and exceptions are generated *after* these pointer data members have been initialized, memory leaks can still be avoided, though. This is accomplished by using *smart pointers*, e.g., *auto_ptr* objects, introduced in section 17.3. As `auto_ptr` objects are objects, their destructors are still called, even when their the full construction of their composing object fails. In this case the rule *once an object has been constructed its destructor is called when the object goes out of scope* still applies.

  Section 17.3.6 covers the use of `auto_ptr` objects to prevent memory leaks when exceptions are thrown out of constructors, even if the exception is generated by a member initializer.

  C++, however, supports an even more generic way to prevent exceptions from leaving functions (or constructors): *function try blocks*. These function try blocks are discussed in the next section.

Destructors have problems of their own when they generate exceptions. Exceptions leaving destructors may of course produce memory leaks, as not all allocated memory may already have been deleted when the exception is generated. Other forms of incomplete handling may be encountered. For example, a database class may store modifications of its database in memory, leaving the update of file containing the database file to its destructor. If the destructor generates an exception before the file has been updated, then there will be no update. But another, far more subtle, consequence of exceptions leaving destructors exist.

The situation we're about to discuss may be compared to a carpenter building a cupboard containing a single drawer. The cupboard is finished, and a customer, buying the cupboard, finds that the cupboard can be used as expected. Satisfied with the cupboard, the customer asks the carpenter to build another cupboard, this time containing *two* drawers. When the second cupboard is finished, the customer takes it home and is utterly amazed when the second cupboard completely collapses immediately after its first use.

Weird story? Consider the following program:

```
int main()
{
    try
    {
        cerr << "Creating Cupboard1\n";
        Cupboard1();
        cerr << "Beyond Cupboard1 object\n";
    }
    catch (...)
    {
        cerr << "Cupboard1 behaves as expected\n";
    }
    try
```

```
        {
            cerr << "Creating Cupboard2\n";
            Cupboard2();
            cerr << "Beyond Cupboard2 object\n";
        }
        catch (...)
        {
            cerr << "Cupboard2 behaves as expected\n";
        }
    }
```

When this program is run it produces the following output:

```
    Creating Cupboard1
    Drawer 1 used
    Cupboard1 behaves as expected
    Creating Cupboard2
    Drawer 2 used
    Drawer 1 used
    Abort
```

The final `Abort` indicating that the program has aborted, instead of displaying a message like `Cupboard2 behaves as expected`. Now let's have a look at the three classes involved. The class `Drawer` has no particular characteristics, except that its destructor throws an exception:

```
    class Drawer
    {
        size_t d_nr;
        public:
            Drawer(size_t nr)
            :
                d_nr(nr)
            {}
            ~Drawer()
            {
                cerr << "Drawer " << d_nr << " used\n";
                throw 0;
            }
    };
```

The class `Cupboard1` has no special characteristics at all. It merely has a single composed `Drawer` object:

```
    class Cupboard1
    {
        Drawer left;
        public:
            Cupboard1()
            :
                left(1)
            {}
    };
```

The class Cupboard2 is constructed comparably, but it has two composed Drawer objects:

```
class Cupboard2
{
    Drawer left;
    Drawer right;
    public:
        Cupboard2()
        :
            left(1),
            right(2)
        {}
};
```

When Cupboard1's destructor is called, Drawer's destructor is eventually called to destroy its composed object. This destructor throws an exception, which is caught beyond the program's first try block. This behavior is completely as expected. However, a problem occurs when Cupboard2's destructor is called. Of its two composed objects, the destructor of the second Drawer is called first. This destructor throws an exception, which ought to be caught beyond the program's second try block. However, although the flow of control by then has left the context of Cupboard2's destructor, that object hasn't completely been destroyed yet as the destructor of its other (left) Drawer still has to be called. Normally that would not be a big problem: once the exception leaving Cupboard2's destructor is thrown, any remaining actions would simply be ignored, albeit that (as both drawers are properly constructed objects) left's destructor would still be called. So this happens here too. However, left's destructor *also* throws an exception. Since we've already left the context of the second try block, the programmed flow control is completely mixed up, and the program has no other option but to abort. It does so by calling terminate(), which in turn calls abort(). Here we have our collapsing cupboard having two drawers, even though the cupboard having one drawer behaves perfectly.

The program aborts since there are multiple composed objects whose destructors throw exceptions leaving the destructors. In this situation one of the composed objects would throw an exception by the time the program's flow control has already left its proper context. This causes the program to abort.

This situation can be prevented if we ensure that exceptions *never* leave destructors. In the cupboard example, Drawer's destructor throws an exception leaving the destructor. This should not happen: the exception should be caught by Drawer's destructor itself. Exceptions should never be thrown out of destructors, as we might not be able to catch, at an outer level, exceptions generated by destructors. As long as we view destructors as service members performing tasks that are *directly* related to the object being destroyed, rather than a member on which we can base any flow control, this should not be a serious limitation. Here is the skeleton of a destructor whose code might throw exceptions:

```
Class::~Class()
{
    try
    {
        maybe_throw_exceptions();
    }
    catch (...)
    {}
}
```

8.9 Function try blocks

Exceptions might be generated while a constructor is initializing its members. How can exceptions generated in such situations be caught by the constructor itself, rather than outside of the constructor? The intuitive solution, nesting the object construction in a nested `try` block does not solve the problem (as the exception by then has left the constructor) and is not a very elegant approach by itself, because of the resulting additional (and somewhat artificial) nesting level.

Using a nested `try` block is illustrated by the next example, where `main()` defines an object of class `DataBase`. Assuming that `DataBase`'s constructor may throw an exception, there is no way we can catch the exception in an 'outer block' (i.e., in the code calling `main()`), as we don't have an outer block in this situation. Consequently, we must resort to less elegant solutions like the following:

```
int main(int argc, char **argv)
{
    try
    {
        DataBase db(argc, argv);    // may throw exceptions
        ...                         // main()'s other code
    }
    catch(...)                      // and/or other handlers
    {
        ...
    }
}
```

This approach may potentially produce very complex code. If multiple objects are defined, or if multiple sources of exceptions are identifiable within the `try` block, we either get a complex series of exception handlers, or we have to use multiple nested `try` blocks, each using its own set of `catch`-handlers.

None of these approaches, however, solves the basic problem: how can exceptions generated in a *local context* be caught before the local context has disappeared?

A function's local context remains accessible when its body is defined as a *function try block*. A function try block consists of a `try` block and its associated handlers, defining the function's body. When a function try block is used, the function itself may catch any exception its code may generate, even if these exceptions are generated in member initializer lists of constructors.

The following example shows how a function try block might have been deployed in the above `main()` function. Note how the `try` block and its handler now replace the plain function body:

```
int main(int argc, char **argv)
try
{
    DataBase db(argc, argv);    // may throw exceptions
    ...                         // main()'s other code
}
catch(...)                      // and/or other handlers
{
    ...
}
```

Of course, this still does not enable us have exceptions thrown by `DataBase`'s constructor itself caught locally by `DataBase`'s constructor. Function try blocks, however, may also be used when implementing constructors. In that case, exceptions thrown by base class initializers (cf. chapter 13) or member initializers may also be caught by the constructor's exception handlers. So let's try to implement this approach.

The following example shows a function try block being used by a constructor. Note that the grammar requires us to put the `try` keyword even before the member initializer list's colon:

```
#include <iostream>

class Throw
{
    public:
        Throw(int value)
        try
        {
            throw value;
        }
        catch(...)
        {
            std::cout << "Throw's exception handled locally by Throw()\n";
            throw;
        }
};

class Composer
{
    Throw d_t;
    public:
        Composer()
        try             // NOTE: try precedes initializer list
        :
            d_t(5)
        {}
        catch(...)
        {
            std::cout << "Composer() caught exception as well\n";
        }
};

int main()
{
    Composer c;
}
```

In this example, the exception thrown by the `Throw` object is first caught by the object itself. Then it is rethrown. As the `Composer`'s constructor uses a function try block, `Throw`'s rethrown exception is also caught by `Composer`'s exception handler, even though the exception was generated inside its member initializer list.

However, when running this example, we're in for a nasty surprise: the program runs and then breaks with an *abort exception*. Here is the output it produces, the last two lines being added by the system's final catch-all handler, catching all exceptions that otherwise remain uncaught:

```
Throw's exception handled locally by Throw()
```

```
Composer() caught exception as well
terminate called after throwing an instance of 'int'
Abort
```

The reason for this is actually stated in the **C++** standard: at the end of a catch-handler implemented as part of a destructor's or constructor's function try block, the original exception is automatically rethrown. The exception is not rethrown if the handler itself throws another exception, and it is not retrown by catch-handlers that are part of try blocks of other functions. Only constructors and destructors are affected. Consequently, to repair the above program another, outer, exception handler is still required. A simple repair (applicable to all programs except those having global objects whose constructors or destructors use function try blocks) is to provide `main` with a function try block. In the above example this would boil down to:

```
int main()
try
{
    Composer c;
}
catch (...)
{}
```

Now the program runs as planned, producing the following output:

```
Throw's exception handled locally by Throw()
Composer() caught exception as well
```

A final note: if a constructor or function using a function try block also declares the exception types it may throw, then the function try block must follow the function's exception specification list.

8.10 Standard Exceptions

All data types may be thrown as exceptions. However, the standard exceptions are derived from the *class exception*. Class derivation is covered in chapter 13, but the concepts that lie behind inheritance are not required for the the current section.

All standard exceptions (and all user-defined classes derived from the class `std::exception`) offer the member

```
char const *what() const;
```

describing in a short textual message the nature of the exception.

Four classes derived from `std::exception` are offered by the language:

- `std::bad_alloc`: thrown when `operator new` fails;
- `std::bad_exception`: thrown when a function tries to generate another type of exception than declared in its function throw list;
- `std::bad_cast`: thrown in the context of *polymorphism* (see section 14.5.1);
- `std::bad_typeid`: also thrown in the context of *polymorphism* (see section 14.5.2);

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 operating on an array of `int`s. 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. Here is an example showing the use of the class:

```
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 `int`s. 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
{
    int     *d_data;
    unsigned d_size;
```

```
    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 const &operator[](unsigned index) const;    // second
    private:
        void boundary(unsigned index) const;
        void copy(IntArray const &other);
        int &operatorIndex(unsigned index) const;
};
```

This class has the following characteristics:

- One of its constructors has an `size_t` parameter having a default argument value, specifying the number of `int` elements in the object.

- The class internally uses a pointer to reach allocated memory. Hence, the necessary tools are provided: a copy constructor, an overloaded assignment operator and a destructor.

- Note that there are two overloaded index operators. Why are there two 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 or 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 `const`, as we must be able to change the element we want to access, when 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 const &`, 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 only used with `const` objects. It is used for value-retrieval, not for value-assignment, but that is precisely what we want, using `const` objects. Here, members are overloaded only by their `const` attribute. This form of function overloading was introduced earlier in the Annotations (sections 2.5.11 and 6.2).

  Also note that, since the values stored in the `IntArray` are primitive values of type `int`, it's ok to use value return types. However, with objects one usually doesn't want the extra copying that's implied with value return types. In those cases `const &` return values are preferred for `const` member functions. So, in the `IntArray` class an `int` return value could have been used as well. The second overloaded index operator would then use the following prototype:

  ```
  int 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.

Now, the implementation of the members are:

```
#include "intarray.ih"

IntArray::IntArray(unsigned size)
:
    d_size(size)
{
    if (d_size < 1)
    {
        cerr << "IntArray: size of array must be >= 1\n";
        exit(1);
    }
    d_data = new int[d_size];
}

IntArray::IntArray(IntArray const &other)
{
    copy(other);
}

IntArray::~IntArray()
{
    delete[] d_data;
}

IntArray const &IntArray::operator=(IntArray const &other)
{
    if (this != &other)
    {
        delete[] d_data;
        copy(other);
    }
    return *this;
}

void IntArray::copy(IntArray const &other)
{
    d_size = other.d_size;
    d_data = new int[d_size];
    memcpy(d_data, other.d_data, d_size * sizeof(int));
}

int &IntArray::operatorIndex(unsigned index) const
{
    boundary(index);
    return d_data[index];
}

int &IntArray::operator[](unsigned index)
{
    return operatorIndex(index);
}
```

```
int const &IntArray::operator[](unsigned index) const
{
    return operatorIndex(index);
}

void IntArray::boundary(unsigned index) const
{
    if (index >= d_size)
    {
        cerr << "IntArray: boundary overflow, index = " <<
                index << ", should range from 0 to " << d_size - 1 << endl;
        exit(1);
    }
}
```

Especially note the implementation of the `operator[]()` functions: as non-const members may call const member functions, and as the implementation of the `const` member function is identical to the non-const member function's implementation, we could implement both `operator[]` members inline using an auxiliary function `int &operatorIndex(size_t index) const`. It is interesting to note that a `const` member function may return a non-const reference (or pointer) return value, referring to one of the data members of its object. This is a potentially dangerous backdoor breaking data hiding. However, as the members in the public interface prevents this breach, we feel confident in defining `int &operatorIndex() const` as a private function, knowing that it won't be used for this unwanted purpose.

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 (<<). Adapting a class in such a way that the `istream`'s extraction operator (>>) can be used, is implemented similarly 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`. Having made 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.name() <<
            "Address: " << pers.address() <<
            "Phone:   " << pers.phone();
}
```

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 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 d_address;
    d_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 (>>). A simple implementation is:

```
istream &operator>>(istream &str, Person &p)
{
    string name;
    string address;
    string 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
{
    char *d_string;

    public:
        String();
        String(char const *arg);
        ~String();
        String(String const &other);
        String const &operator=(String const &rvalue);
        String const &operator=(char const *rvalue);
};
```

Objects 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 *`[1].

Usually, in classes that are less directly coupled 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
{
    String *d_store;
    size_t d_n;

    public:
        StringArray(size_t size);
        StringArray(StringArray const &other);
        StringArray const &operator=(StringArray const &rvalue);
        ~StringArray();

        String &operator[](size_t index);
};
```

Using the `StringArray::operator[]`, assignments between the `String` elements can simply be realized:

[1]Note that the assingment from a `char const *` also includes the null-pointer. An assignment like `stringObject = 0` is perfectly in order.

```
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";
```

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 9.11). 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 d_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 produces a compilation error:

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

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.

Several considerations apply to conversion operators:

- In general, a class should have at most one conversion operator. When multiple conversion operators are defined, ambiguities are quickly introduced.
- A conversion operator should be a 'natural extension' of the facilities of the object. For example, the stream classes define `operator bool()`, allowing constructions like `if (cin)`.

- A conversion operator should return a rvalue. It should do so not only to enforce data-hiding, but also because implementing a conversion operator as an lvalue simply won't work. The following little program is a case in point: the compiler will not perform a two-step conversion and will therefore try (in vain) to find `operator=(int)`:

```
#include <iostream>

class Lvalue
{
    int d_value;

    public:
        operator int&();
};

    inline Lvalue::operator int&()
    {
        return d_value;
    }

int main()
{
    Lvalue lvalue;

    lvalue = 5;     // won't compile: no lvalue::operator=(int)
};
```

- Conversion operators should be defined as `const` member functions if they don't modify their object's data members.
- Conversion operators returning composed objects should return const references to these objects, rather than the plain object types. Plain object types would force the compiler to call the composed object's copy constructor, instead of a reference to the object itself. For example, in the following program `std::string`'s copy constructor is not called. It would have been called if the conversion operator had been declared as `operator string()`:

```
#include <string>

class XString
{
    std::string d_s;

    public:
        operator std::string const &() const;
};

inline XString::operator std::string const &() const
{
    return d_s;
}

int main()
{
    XString x;
    std::string s;
```

```
        s = x;
    };
```

9.4 The keyword ‘explicit’

Conversions are performed not only by conversion operators, but also by constructors having one parameter (or multiple parameters, having default argument 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 argument 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 113", "050 363 9281");
```

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 abovementioned `Person` constructor: the second and third parameters use their default values. Here, an *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 to allow for the new table size.

Such a class could (partially) be defined as follows:

```
class HashTable
{
    size_t d_maxsize;

    public:
        HashTable(size_t n);  // n: initial table size
        size_t size();        // returns current # of elements

                                // add new key and value
        void add(std::string const &key, std::string const &value);
};
```

Now consider the following implementation of `add()`:

```
void HashTable::add(string const &key, string const &value)
{
    if (size() > d_maxsize * 0.75)  // table gets rather full
        *this = 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. Although this succeeds, the hashtable will completely fail to fulfill its purpose: accidentally the programmer assigns an size_t value, intending to tell the hashtable what its new size should be. This results in the following unwelcome surprise:

- The compiler notices that no `operator=(size_t newsize)` is available for `HashTable`.
- There is, however, a constructor accepting an size_t, *and* the default overloaded assignment operator is still available, expecting a `HashTable` as its right-hand operand.
- Thus, the rvalue of the assignment (a `HashTable`) is obtained by (implicitly) constructing an (empty) `HashTable` that can accomodate `size() * 2` elements.
- The just constructed empty `HashTable` is thereupon 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 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 `size_t` to `HashTable` the class interface of the class `HashTable` should declare the constructor

```
explicit HashTable(size_t n);
```

Now the compiler will catch the error in the compilation of `HashTable::add()`, producing an error message like

```
error: no match for 'operator=' in
            '*this = (this->HashTable::size()() * 2)'
```

9.5 Overloading the increment and decrement operators

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*, which is an expression having 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 `size_t` value type. The class could have the following (partially shown) interface:

```
class Unsigned
{
    size_t d_value;

    public:
        Unsigned();
        Unsigned(size_t init);
        Unsigned &operator++();
}
```

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++()
{
    ++d_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 an acceptable application of function overloading. Whatever your opinion in this matter, the following can be concluded:

- Overloaded increment and decrement operators *without parameters* are *prefix* operators, and should return *references* to the current object.
- Overloaded increment and decrement operators *having an int parameter* are *postfix* operators, and should return the value the object has at the point the overloaded operator is called as a constant value.

To add the postfix increment operator to the `Unsigned` wrapper class, add the following line to the class interface:

```
Unsigned const operator++(int);
```

The *implementation* of the postfix increment operator should be like this:

```
Unsigned const Unsigned::operator++(int)
{
    return d_value++;
}
```

The simplicity of this implementation is *deceiving*. Note that:

- `d_value` is used with a postfix increment in the `return` expression. Therefore, the value of the `return` expression is `d_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 `d_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, using a copy constructor might be preferred. 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, 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 initializing 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 an (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 binary operators

In various classes overloading binary operators (like `operator+()`) can be a very natural extension of the class’s functionality. For example, the `std::string` class has various overloaded forms of `operator+()` as have most *abstract containers*, covered in chapter 12.

Most binary operators come in two flavors: the plain binary operator (like the + operator) and the arithmetic assignment variant (like the += operator). Whereas the plain binary operators return const expression values, the arithmetic assignment operators return a (non-const) reference to the object to which the operator was applied. For example, with `std::string` objects the following code (annotated below the example) may be used:

```
std::string s1;
std::string s2;
std::string s3;

s1 = s2 += s3;                  // 1
(s2 += s3) + " postfix";        // 2
s1 = "prefix " + s3;            // 3
"prefix " + s3 + "postfix";     // 4
("prefix " + s3) += "postfix";  // 5
```

- at `// 1` the contents of `s3` is added to `s2`. Next, `s2` is returned, and its new contents are assigned to `s1`. Note that += returns `s2` itself.
- at `// 2` the contents of `s3` is also added to `s2`, but as += returns `s2` itself, it’s possible to add some more to `s2`
- at `// 3` the + operator returns a `std::string` containing the concatenation of the text `prefix` and the contents of `s3`. This string returned by the + operator is thereupon assigned to `s1`.

- at `// 4` the + operator is applied twice. The effect is:
    1. The first + returns a `std::string` containing the concatenation of the text `prefix` and the contents of `s3`.
    2. The second + operator takes this returned string as its left hand value, and returns a string containing the concatenated text of its left and right hand operands.
    3. The string returned by the second + operator represents the value of the expression.
- statement `// 5` should not compile (although it does compile with the `Gnu` compiler version 3.1.1). It should not compile, as the + operator should return a `const` string, thereby preventing its modification by the subsequent += operator. Below we will consequently follow this line of reasoning, and will ensure that overloaded binary operators will always return `const` values.

Now consider the following code, in which a class `Binary` supports an overloaded `operator+()`:

```
class Binary
{
    public:
        Binary();
        Binary(int value);
        Binary const operator+(Binary const &rvalue);
};

int main()
{
    Binary b1;
    Binary b2(5);

    b1 = b2 + 3;            // 1
    b1 = 3 + b2;            // 2
}
```

Compilation of this little program fails for statement `// 2`, with the compiler reporting an error like:

```
error: no match for 'operator+' in '3 + b2'
```

Why is statement `// 1` compiled correctly whereas statement `// 2` won't compile?

In order to understand this, the notion of a *promotion* is introduced. As we have seen in section 9.4, constructors requiring a single argument may be implicitly activated when an object is apparently initialized by an argument of a corresponding type. We've encountered this repeatedly with `std::string` objects, when an `ASCII-Z` string was used to initialize a `std::string` object.

In situations where a member function expects a `const &` to an object of its own class (like the `Binary const &` that was specified in the declaration of the `Binary::operator+()` member mentioned above), the type of the actually used argument may also be any type that can be used as an argument for a single-argument constructor of that class. This implicit call of a constructor to obtain an object of the proper type is called a *promotion*.

So, in statement `// 1`, the + operator is called for the `b2` object. This operator expects another `Binary` object as its right hand operand. However, an `int` is provided. As a constructor `Binary(int)` exists, the `int` value is first promoted to a `Binary` object. Next, this `Binary` object is passed as argument to the `operator+()` member.

Note that no promotions are possibly in statement `// 2`: here the + operator is applied to an `int` typed value, which has no concept of a 'constructor', 'member function' or 'promotion'.

How, then, are promotions of left-hand operands realized in statements like `"prefix " + s3`? Since promotions are applied to function arguments, we must make sure that both operands of binary operators are arguments. This means that binary operators are declared as *classless functions*, also called *free functions*. However, they conceptually belong to the class for which they implement the binary operator, and so they should be declared in the class's header file. We will cover their implementations shortly, but here is our first revision of the declaration of the class `Binary`, declaring an overloaded + operator as a free function:

```
class Binary
{
    public:
        Binary();
        Binary(int value);
};

Binary const operator+(Binary const &l_hand, Binary const &r_hand);
```

By defining binary operators as free functions, the following promotions are possible:

- If the left-hand operand is of the intended class type, the right hand argument will be promoted whenever possible
- If the right-hand operand is of the intended class type, the left hand argument will be promoted whenever possible
- No promotions occur when none of the operands are of the intended class type
- An ambiguity occurs when promotions to different classes are possible for the two operands. For example:

```
class A;

class B
{
    public:
        B(A const &a);
};

class A
{
    public:
        A();
        A(B const &b);
};

A const operator+(A const &a, B const &b);
B const operator+(B const &b, A const &a);

int main()
```

```
    {
        A a;

        a + a;
    };
```

Here, both overloaded + operators are possible when compiling the statement `a + a`. The ambiguity must be solved by explicitly promoting one of the arguments, e.g., `a + B(a)` will allow the compiler to resolve the ambiguity to the first overloaded + operator.

The next step is to implement the corresponding overloaded arithmetic assignment operator. As this operator *always* has a left-hand operand which is an object of its own class, it is implemented as a true member function. Furthermore, the arithmetic assignment operator should return a reference to the object to which the arithmetic operation applies, as the object might be modified in the same statement. E.g., `(s2 += s3) + " postfix"`. Here is our second revision of the class `Binary`, showing both the declaration of the plain binary operator and the corresponding arithmetic assignment operator:

```
class Binary
{
    public:
        Binary();
        Binary(int value);
        Binary const operator+(Binary const &rvalue);

        Binary &operator+=(Binary const &other);
};

Binary const operator+(Binary const &l_hand, Binary const &r_hand);
```

Finally, having available the arithmetic assignment operator, the implementation of the plain binary operator turns out to be extremely simple. It contains of a single return statement, in which an anonymous object is constructed to which the arithmetic assignment operator is applied. This anonymous object is then returned by the plain binary operator as its `const` return value. Since its implementation consists of merely one statement it is usually provided in-line, adding to its efficiency:

```
class Binary
{
    public:
        Binary();
        Binary(int value);
        Binary const operator+(Binary const &rvalue);

        Binary &operator+=(Binary const &other);
};

Binary const operator+(Binary const &l_hand, Binary const &r_hand)
{
    return Binary(l_hand) += r_hand;
}
```

One might wonder where the temporary value is located. Most compilers apply in these cases a procedure called '*return value optimization*': the anonymous object is created at the location where the eventual returned object will be stored. So, rather than first creating a separate temporary object, and then copying this object later on to the return value, it initializes the return value using the `l_hand` argument, and then applies the `+=` operator to add the `r_hand` argument to it. Without return value optimization it would have to:

- create separate room to accomodate the return value
- initialize a temporary object using `l_hand`
- Add `r_hand` to it
- Use the copy constructor to copy the temporary object to the return value.

Return value optimization is not required, but optionally available to compilers. As it has no negative side effects, most compiler use it.

9.7 Overloading 'operator new(size_t)'

When `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 9.9. 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[2]:

```
class X
{
    public:
        void *operator new(size_t sizeofX);

        int d_x;
        int d_y;
};
```

[2]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>
using namespace std;

int main()
{
    X *x = new X();

    cout << x->d_x << ", " << x->d_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 for constructors. 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 *ptr = X::operator new();  // new allocates the memory

X::X(ptr);                        // next the constructor operates 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` 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 (as the `this` pointer) to the constructor 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, size_t 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 *`.

Finally note that no explicit argument is passed for `new`'s first parameter, as this argument is implicitly provided by the type specification that's required for operator `new`.

9.8 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 9.9.

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;
}
```

9.9 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's 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 wil 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.

9.9.1 Overloading 'new[]'

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, size_t 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)]
```

Or, alternatively, the required (uninitialized) amount of memory can be allocated using:

```
::new char[size]
```

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 additional parameters. The second form of the overloaded operator `new[]` shows such an additional `size_t` parameter. The definition of such a function is standard, and could be:

```
void *Object::operator new[](size_t size, size_t extra)
{
    size_t n = size / sizeof(Object);
```

```
        Object *op = ::new Object[n];

        for (size_t idx = 0; idx < n; idx++)
            op[idx].value = extra;          // assume a member 'value'

        return op;
    }
```

To use this overloaded operator, only the additional 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` members set to 100.

9.9.2 Overloading 'delete[]'

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);
    };
```

9.9.2.1 'delete[](void *)'

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 `::delete[]`. 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];
```

9.9.2.2 'delete[](void *, size_t)'

Operator `delete[]` may be overloaded using additional 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, size_t size)
{
    cout << "deleting " << size << " bytes\n";
    ::delete[] reinterpret_cast<Object *>(p);
}
```

9.9.2.3 Alternate forms of overloading operator 'delete[]'

If additional 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 *following* the brackets:

```
delete[](new Object[5], 100, false);
```

9.10 Function Objects

Function Objects are created by overloading the *function call operator* `operator()()`. By defining the function call operator an object masquerades as a function, hence the term *function objects*.

Function objects play an important role in *generic algorithms* and their use is preferred over alternatives like pointers to functions. The fact that they are important in the context of generic algorithms constitutes some sort of a didactical dilemma: at this point it would have been nice if generic algorithms would have been covered, but for the discussion of the generic algorithms knowledge of function objects is required. This bootstrapping problem is solved in a well known way: by ignoring the dependency.

Function objects are objects for which `operator()()` has been defined. Function objects are commonly used in combination with generic algorithms, but also 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 *lineair search* in an array. A program fragment using this function is:

```
Person &target = targetPerson();    // determine the person to find
```

```
Person *pArray;
size_t n = fillPerson(&pArray);

cout << "The target person is";

if (!lsearch(&target, pArray, &n, sizeof(Person), compareFunction))
    cout << " not";

cout << "found\n";
```

The function `targetPerson()` 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 bytewise 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 average `n / 2` times *at least* the following actions take place:

1. The two arguments of the compare function are pushed on the stack;
2. The value of the final parameter of `lsearch()` is determined, producing the address of `compareFunction()`;
3. The compare function is called;
4. Then, inside the compare function the address of the right-hand argument of the `Person::operator!=()` argument is pushed on the stack;
5. The `Person::operator!=()` function is evaluated;
6. The argument of the `Person::operator!=()` function is popped off the stack again;
7. The two arguments of the compare function are popped off the stack again.

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();
```

```
Person *pArray;
size_t 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:

```
bool 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, than 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 annihilate 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 access the private data of their objects directly. In 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.

As noted, function objects play a central role in generic algorithms. In chapter 17 these generic algorithms are discussed in detail. Furthermore, in that chapter *predefined function objects* will be introduced, further emphasizing the importance of the function object concept.

9.10.1 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 <ostream>
#include <iomanip>

std::ostream &w10(std::ostream &str)
{
    return str << std::setw(10);
}
```

The `w10` function can of course be used in a 'stand alone' mode, but it can also be used as a manipulator. E.g.,

```
#include <iostream>
#include <iomanip>

using namespace std;

extern ostream &w10(ostream &str);

int main()
```

```
    {
        w10(cout) << 3 << " ships sailed to America" << endl;
        cout << "And " << w10 << 3 << " more ships sailed too." << endl;
    }
```

The `w10` function can be used as a manipulator because the `class ostream` has an overloaded `operator<<()` accepting a pointer to a function expecting an `ostream &` and returning an `ostream &`. Its definition is:

```
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. In **C++** programs they should be avoided whenever possible. The following section introduces a way to implement manipulators requiring arguments without resorting to macros, but using anonymous objects.

9.10.1.1 Manipulators requiring arguments

Manipulators taking arguments are implemented as macros: they are handled by the preprocessor, and are not available beyond the preprocessing stage. The problem appears to be that you can't call a function in an insertion sequence: in a sequence of `operator<<()` calls the compiler will first call the functions, and then use their return values in the insertion sequence. That will invalidate the ordering of the arguments passed to your <<-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. `Align`, whose constructor expects multiple arguments. In our example representing, respectively, the field width and the alignment.
- Furthermore, we define the function:

  ```
  ostream &operator<<(ostream &ostr, Align const &align)
  ```

  so we can insert an `Align` object into the ostream.

Here is an example of a little program using such a *home-made* manipulator expecting multiple arguments:

```
#include <iostream>
#include <iomanip>

class Align
{
    unsigned d_width;
    std::ios::fmtflags d_alignment;

    public:
        Align(unsigned width, std::ios::fmtflags alignment);
        std::ostream &operator()(std::ostream &ostr) const;
};

    Align::Align(unsigned width, std::ios::fmtflags alignment)
    :
        d_width(width),
        d_alignment(alignment)
    {}

    std::ostream &Align::operator()(std::ostream &ostr) const
    {
        ostr.setf(d_alignment, std::ios::adjustfield);
        return ostr << std::setw(d_width);
    }

std::ostream &operator<<(std::ostream &ostr, Align const &align)
{
    return align(ostr);
}

using namespace std;

int main()
{
    cout
        << "‘" << Align(5, ios::left) << "hi" << "’"
        << "‘" << Align(10, ios::right) << "there" << "’" << endl;
}

/*
    Generated output:

    ‘hi   ’‘     there’
*/
```

Note that in order to insert an anonymous `Align` object into the `ostream`, the `operator<<()` function *must* define a `Align const &` parameter (note the `const` modifier).

9.11 Overloadable operators

The following operators can be overloaded:

```
+       -       *       /       %       ^       &       |
~       !       ,       =       <       >       <=      >=
++      --      <<      >>      ==      !=      &&      ||
+=      -=      *=      /=      %=      ^=      &=      |=
<<=     >>=     []      ()      ->      ->*     new     new[]
delete  delete[]
```

When 'textual' alternatives of operators are available (e.g., `and` for &&) then they are overloadable too.

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.

Finally, the following operators are not overloadable at all:

```
.       .*      ::      ?:      sizeof  typeid
```

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` member could access any member of any object of its 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. For example, the name of the startup directory, used by a program that recursively scans the directory tree of a disk. A second example is a 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 violates our philosophy of using only one function per source file. Another **C**-solution is to give the variable in question an unusual name, e.g., `_6uldv8`, hoping that other program parts won't use this name by accident. Neither the first, nor the second **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 static members are the topic of this chapter.

10.1 Static data

Any data member of a class can be declared `static`; be it in the `public` or `private` section 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. Note, however, that they are always created as true members of their classes. It is suggested to prefix static member names with `s_` in order to distinguish them (in class member functions) from the class's data members (which should preferably start with `d_`).

Public static data members are like 'normal' global variables: they can be accessed by *all code of the program*, simply using their class names, the scope resolution operator and their member names. This is illustrated in the following example:

```
class Test
{
```

```
        static int s_private_int;

        public:
            static int s_public_int;
    };

    int main()
    {
        Test::s_public_int = 145;   // ok

        Test::s_private_int = 12;   // wrong, don't touch
                                    // the private parts
        return 0;
    }
```

This code fragment is not suitable for consumption by a **C++** compiler: it merely 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
{
    static char s_path[];

    public:
        // constructors, destructors, etc. (not shown)
};
```

The data member `s_path[]` is a *private static data member*. During the execution of the program, only *one* `Directory::s_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 static data members exist *before* any constructor of the class has been called. Static data members are initialized when they are defined, outside of all member functions, in the same way as other 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 `s_path[]`, used above, could thus be defined and initialized as follows in a file `data.cc`:

```
include "directory.ih"

char Directory::s_path[200] = "/usr/local";
```

In the class interface the static member is actually only *declared*. In its implementation (definition) its type and class name are explicitly mentioned. Note also that the size specification can be left out of the interface, as shown above. However, its size *is* (either explicitly or implicitly) required when it is defined.

Note that *any* source file could contain the definition of the static data members of a class. A separate `data.cc` source is advised, but the source file containing, e.g., `main()` could be used as well. Of course, any source file defining static data of a class must also include the header file of that class, in order for the static data member to be known to the compiler.

A second example of a useful private static data member is given below. Assume that a class `Graphics` defines the communication of a program with a graphics-capable device (e.g., a VGA screen). The initialization of the device, which in this case would be to switch from text mode to graphics mode, is an action of the constructor and depends on a `static` flag variable `s_nobjects`. The variable `s_nobjects` simply counts the number of `Graphics` objects which are present at one time. Similarly, the destructor of the class may switch back from graphics mode to text mode when the last `Graphics` object ceases to exist. The class interface for this `Graphics` class might be:

```
class Graphics
{
    static int s_nobjects;              // counts # of objects

    public:
        Graphics();
        ~Graphics();                    // other members not shown.
    private:
        void setgraphicsmode();         // switch to graphics mode
        void settextmode();             // switch to text-mode
}
```

The purpose of the variable `s_nobjects` is to count the number of objects existing at a particular moment in time. When the first object is created, the graphics device is initialized. At the destruction of the last `Graphics` object, the switch from graphics mode to text mode is made:

```
int Graphics::s_nobjects = 0;           // the static data member

Graphics::Graphics()
{
    if (!s_nobjects++)
        setgraphicsmode();
}

Graphics::~Graphics()
{
    if (!--s_nobjects)
        settextmode();
}
```

Obviously, when the class `Graphics` would define more than one constructor, each constructor would need to increase the variable `s_nobjects` and would possibly have to initialize the graphics mode.

10.1.2 Public static data

Data members can be declared in the public section of a class, although this is not common practice (as this would violate the principle of data hiding). E.g., when the static data member `s_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::s_path, 199);
}
```

Note that the variable `s_path` would still have to be defined. As before, the class interface would only *declare* the array `s_path[]`. This means that some source file would still need to contain the definition of the `s_path[]` array.

10.1.3 Initializing static const data

Static `const` data members may be initialized in the class interface if these data members are of an integral data type. So, in the following example the first three static data members can be initialized since `int enum` and `double` types are integral data members. The last static data member cannot be initialized in the class interface since `string` is not an integral data type:

```
class X
{
    public:
        enum Enum
        {
            FIRST,
        };

        static int const s_x = 34;
        static Enum const s_type = FIRST;

        static double const s_d = 1.2;
        static string const s_str = "a";    // won't compile
};
```

Static `const` integral data members initialized in the class interface are not addressable variables. They are mere symbolic names for their associated values. Since they are not variables, it is not possible to determine their addresses. Note that this is not a compilation problem, but a linking problem. The static `const` variable that is initialized in the class interface does not exist as an addressable entity.

A statement like `int *ip = &X::s_x` will therefore *compile* correctly, but will fail to *link*. Static variables that are explicitly defined in a source file *can* be linked correctly, though. So, in the following example the address of `X::s_x` cannot be solved by the linker, but the address of `X::s_y` *can* be solved by the linker:

```
class X
{
    public:
        static int const s_x = 34;
        static int const s_y;
};
```

```
int const X::s_y = 12;

int main()
{
    int const *ip = &X::s_x;    // compiles, but fails to link
    ip = &X::s_y;               // compiles and links correctly
}
```

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 do not have a `this` pointer. In fact, a static member function is completely comparable to a global function, not associated with any class (i.e., in practice they are. See the next section (10.2.1) for a subtle note). Since static member functions do not require an associated object, static member functions declared in the public section of a class interface may be called without specifying an object of its class. The following example illustrates this characteristic of static member functions:

```
class Directory
{
    string d_currentPath;
    static char s_path[];

    public:
        static void setpath(char const *newpath);
        static void preset(Directory &dir, char const *path);
};
inline void Directory::preset(Directory &dir, char const *newpath)
{
                                                // see the text below
    dir.d_currentPath = newpath;                // 1
}

char Directory::s_path[200] = "/usr/local";     // 2

void Directory::setpath(char const *newpath)
{
    if (strlen(newpath) >= 200)
        throw "newpath too long";

    strcpy(s_path, newpath);                    // 3
}

int main()
{
    Directory dir;
```

```
    Directory::setpath("/etc");                 // 4
    dir.setpath("/etc");                        // 5

    Directory::preset(dir, "/usr/local/bin");   // 6
    dir.preset(dir, "/usr/local/bin");          // 7
}
```

- at 1 a static member function modifies a private data member of an object. However, the object whose member must be modified is given to the member function as a reference parameter.

  Note that static member functions can be defined as inline functions.
- at 2 a relatively long array is defined to be able to accomodate long paths. Alternatively, a `string` or a pointer to dynamic memory could have been used.
- at 3 a (possibly longer, but not too long) new pathname is stored in the static data member `s_path[]`. Note that here only static members are used.
- at 4, `setpath()` is called. It is a static member, so no object is required. But the compiler must know to which class the function belongs, so the class is mentioned, using the scope resolution operator.
- at 5, the same is 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 the definition of private static member functions: these functions can only be called by member functions of their class.

10.2.1 Calling conventions

As noted in the previous section, static (public) member functions are comparable to classless functions. However, formally this statement is not true, as the **C++** standard does not prescribe the same calling conventions for static member functions and for classless global functions.

In practice these calling conventions are identical, implying that the address of a static member function could be used as an argument in functions having parameters that are pointers to (global) functions.

If unpleasant surprises must be avoided at all cost, it is suggested to create global classless *wrapper functions* around static member functions that must be used as *call back* functions for other functions.

Recognizing that the traditional situations in which call back functions are used in **C** are tackled in **C++** using template algorithms (cf. chapter 17), let's assume that we have a class `Person` having data members representing the person's name, address, phone and weight. Furthermore, assume we want to sort an array of pointers to `Person` objects, by comparing the `Person` objects these pointers point to. To keep things simple, we assume that a public static

```
int Person::compare(Person const *const *p1, Person const *const *p2);
```

exists. A useful characteristic of this member is that it may directly inspect the required data members of the two `Person` objects passed to the member function using double pointers.

Most compilers will allow us to pass this function's address as the address of the comparison function for the standard **C** `qsort()` function. E.g.,

```
qsort
(
    personArray, nPersons, sizeof(Person *),
    reinterpret_cast<int(*)(const void *, const void *)>(Person::compare)
);
```

However, if the compiler uses different calling conventions for static members and for classless functions, this might not work. In such a case, a classless wrapper function like the following may be used profitably:

```
int compareWrapper(void const *p1, void const *p2)
{
    return
        Person::compare
        (
            reinterpret_cast<Person const *const *>(p1),
            reinterpret_cast<Person const *const *>(p2)
        );
}
```

resulting in the following call of the `qsort()` function:

```
qsort(personArray, nPersons, sizeof(Person *), compareWrapper);
```

Note:

- The wrapper function takes care of any mismatch in the calling conventions of static member functions and classless functions;
- The wrapper function handles the required type casts;
- The wrapper function might perform small additional services (like dereferencing pointers if the static member function expects references to `Person` objects rather than double pointers);
- As noted before: in current **C++** programs functions like `qsort()`, requiring the specification of call back functions are seldomly used, in favor of existing generic template algorithms (cf. chapter 17).

Chapter 11

Friends

In all examples we've discussed up to now, we've seen that `private` members are only accessible by the members of their class. This is *good*, as it enforces 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 to allow 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::name()`. 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 members, 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
{
    Person *d_person;
    size_t d_n;
};
```

When constructing an overloaded insertion operator for a PersonData object, e.g., inserting the information of all its persons into a stream, the overloaded insertion operator is implemented rather inefficiently when the individual persons must be accessed using the index operator.

In cases like these, where the accessor and modifier members tend to become rather complex, direct access to the private data members might improve efficiency. So, in the context of insertion and extraction, we are looking for overloaded member functions implementing the insertion and extraction operations and having 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. The friend keyword is used to realize this.

11.1 Friend functions

Concentrating on the PersonData class, our initial implementation of the insertion operator is:

```
ostream &operator<<(ostream &str, PersonData const &pd)
{
    for (size_t 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, the information about every Person stored in the PersonData object will be written on a separate line.

However, repeatedly calling the index operator might reduce the efficiency of the implementation. Instead, directly using the array Person *d_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 sensibly 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* refer neither to `private` nor to `public` functions, so the friend declaration may be placed anywhere in the class interface. Convention dictates that friend declaractions 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 so as to allow the insertion operator direct access to the private data members of the provided `PersonData` object:

```
ostream &operator<<(ostream &str, PersonData const &pd)
{
    for (size_t idx = 0; idx < pd.d_n; idx++)
        str << pd.d_person[idx] << endl;
}
```

Once again, whether `friend` functions are considered acceptable or not remains a matter of taste: if the function is in fact considered a member function, but it cannot be defined as a member function due to the nature of the **C++** grammar, then it is defensible to use the `friend` keyword. In other cases, the `friend` keyword should rather 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 extend this line of reasoning a little further.

If we conceptually consider friend functions to be member functions, 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 (size_t idx = 0; idx < d_n; idx++)
        str << d_person[idx] << endl;
    return str;
}
```

This member function can be used by 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 insertion operator, earlier defined as a `friend`, we could simply call the `insertor()` member in the code of the `friend` `operator<<()` function. Now this `operator<<()` function needs *only one statement*: it 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* member, as it contains but one statement. However, this is deprecated since it contaminates class interfaces with implementations. The overloaded `operator<<()` member should be implemented below the class interface:

Thus, the relevant section of the class interface of `PersonData` becomes:

```
class PersonData
{
    friend ostream &operator<<(ostream &str, PersonData const &pd);

    private:
        ostream &insertor(ostream &str) const;
};

inline std::ostream &operator<<(std::ostream &str, PersonData const &pd)
{
    return pd.insertor(str);
}
```

The above example illustrates 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 by 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 operates. As the data types 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 *angle bracket notation* is used to indicate which data types are required. For example, below (in section 12.2) we'll encounter the `pair` container, which requires the explicit mentioning of two data types. 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 angle 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 abstract containers. Now that we've introduced this notation, we can postpone the more thorough discussion of templates to chapter 18, and concentrate on their use in this chapter.

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 storing keys and their associated values. Elements of these maps are returned as `pairs`.

A variant of the `pair` is the `complex` container, implementing operations that are defined on *complex numbers*.

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 ANSI/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 types 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 side container is less than each corresponding element in the right-hand side container. Additional elements in either the left-hand side container or the right-hand side container are ignored.

```
container left;
container right;

left = {0, 2, 4};
right = {1, 3};             // left < right

right = {1, 3, 6, 1, 2};    // left < right
```

Note that before a user-defined type (usually a `class`-type) can be stored in a container, the user-defined type should at least support:

- A default-value (e.g., a default constructor)
- The equality operator (==)
- The less-than operator (<)

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 usually support constructors and members expecting iterators, and they often have members returning iterators (comparable to the `string::begin()` and `string::end()`

members). In the remainder of this chapter the iterator concept is not covered. Refer to chapter 17 for this.

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 lifetimes. When a container goes out of scope, its destructor tries to destroy its data elements. This only succeeds if the data elements themselves are stored inside the container. If the data elements of containers are pointers, the data pointed to by these pointers will not be destroyed, resulting in a memory leak. 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 container's destructor is called. So, normally containers should contain no pointer data. Also, a container should not be required to contain `const` data, as `const` data prevent the use of many of the container's members, like the assignment operator.

12.1 Notations used in this chapter

In this chapter about containers, the following notational convention is used:

- Containers live in the standard namespace. In code examples this will be clearly visible, but in the text `std::` is usually omitted.
- A container without angle brackets represents any container of that type. Mentally add the required type in angle bracket notation. E.g., `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 pairs 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.

12.2 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. Before `pair` containers can be used the following preprocessor directive must have been specified:

```
#include <utility>
```

The data types of a `pair` are specified when the `pair` variable is defined (or declared), using the standard template (see chapter `Templates`) angle bracket notation:

```
pair<string, string> piper("PA28", "PH-ANI");
pair<string, string> 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 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) angle bracket notation is used again. E.g., the reassignment of the `cessna` pair variable could have been accomplished as follows:

```
cessna = pair<string, string>("C152", "PH-ANW");
```

In cases like these, the `type` specification can become quite elaborate, which has caused a revival of interest in the possibilities offered by the `typedef` keyword. If 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`.

12.3 Sequential Containers

12.3.1 The 'vector' container

The `vector` class implements an expandable array. Before `vector` containers can be used the following preprocessor directive must have been specified:

```
#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 angle brackets, just after the '`vector`' container name. This is common practice with containers.
  - A vector may be initialized to 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. Some examples:

    ```
    vector<string> object(5, string("Hello")); // initialize to 5 Hello's,
    vector<string> container(10);              // and to 10 empty strings
    ```
  - A vector may be initialized using iterators. To initialize a vector with elements 5 until 10 (including the last one) of an existing `vector<string>` the following construction may be used:

    ```
    extern vector<string> container;
    vector<string> object(&container[5], &container[11]);
    ```

    Note here that the last element pointed to by the second iterator (`&container[11]`) is *not* stored in `object`. This is a simple example of the use of *iterators*, in which the range of values that is used starts at the first value, and includes all elements up to but not including the element to which the second iterator refers. The standard notation for this is `[begin, end)`.
  - A vector may be initialized using a copy constructor:

    ```
    extern vector<string> container;
    vector<string> object(container);
    ```

- In addition to 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 to use the member only if the vector is not empty.
  - `vector::iterator vector::begin()`:

    this member returns an iterator pointing to the first element in the vector, returning `vector::end()` if the vector is empty.
  - `vector::clear()`:

    this member erases all the vector's elements.

- `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 the iterator `pos`. The value `++pos` is returned.
  * `erase(first, beyond)` erases elements indicated by the iterator range `[first, beyond)`, returning `beyond`.
- `Type &vector::front()`:

  this member returns a reference to the first element in the vector. It is the responsibility of the programmer to use the member only if the vector is not empty.
- `... 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 `value` is not provided, the additional elements are initialized to the default value of the used data type, otherwise `value` is used to initialize extra elements.
- `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.
- `size_t vector::size()`

  this member returns the number of elements in the vector.
- `void vector::swap()`

  this member can be used to swap two vectors using identical data types. E.g.,

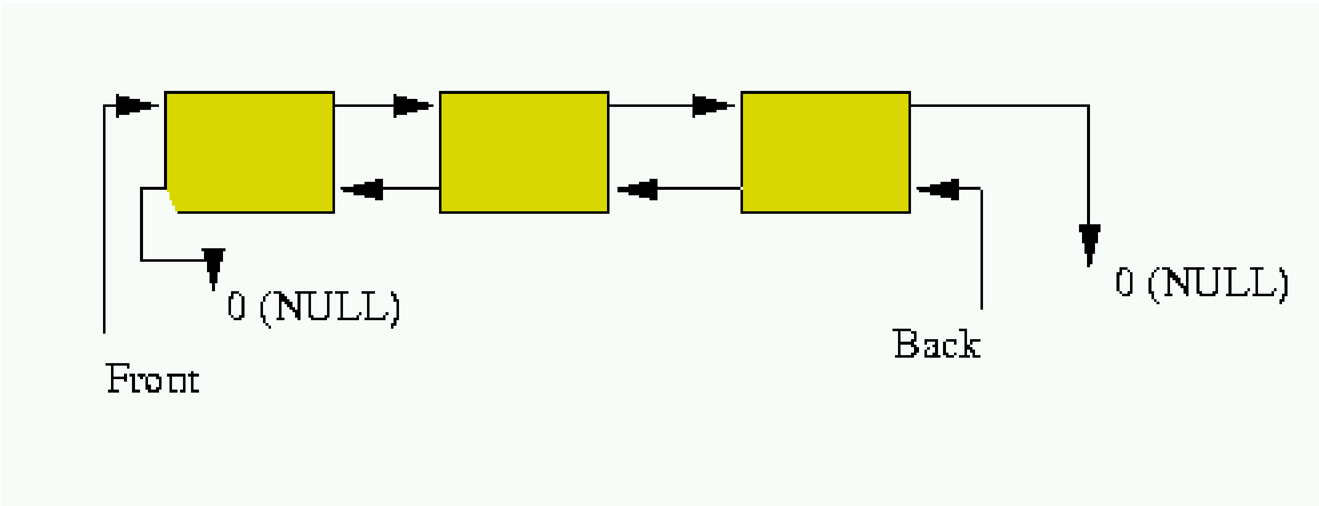

Figure 12.1: A list data-structure

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

int main()
{
    vector<int> v1(7);
    vector<int> v2(10);

    v1.swap(v2);
    cout << v1.size() << " " << v2.size() << endl;
}
/*
    Produced output:
10 7
*/
```

12.3.2 The 'list' container

The `list` container implements a list data structure. Before `list` containers can be used the following preprocessor directive must have been specified:

```
#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 directions: starting at *Front* the list may be traversed from left to right, until the 0-pointer is reached at the end of the rightmost list-element. The list can also be traversed from right to left: starting at *Back*, the list is traversed from right to left, until eventually the 0-pointer emanating from the leftmost list-element is reached.

As a subtlety note that the representation given in figure 12.1 is not necessarily used in actual implementations of the list. For example, consider the following little program:

```
int main()
{
    list<int> l;
    cout << "size: " << l.size() << ", first element: " <<
            l.front() << endl;
}
```

When this program is run it might actually produce the output:

```
size: 0, first element: 0
```

Its front element can even be assigned a value. In this case the implementor has choosen to insert a hidden element to the list, which is actually a circular list, where the hidden element serves as terminating element, replacing the 0-pointers in figure 12.1. As noted, this is a subtlety, which doesn't affect the conceptual notion of a list as a data structure ending in 0-pointers. Note also that it is well known that various implementations of list-structures are possible (cf. Aho, A.V., Hopcroft J.E. and Ullman, J.D., (1983) *Data Structures and Algorithms* (Addison-Wesley)).

Both lists and vectors are often appropriate data structures in situations where an unknown number of data elements must be stored. However, there are some rules of thumb to follow when a choice between the two data structures must be made.

- When the majority of accesses is random, a `vector` is the preferred data structure. E.g., a program counting 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.

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 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;
    ```

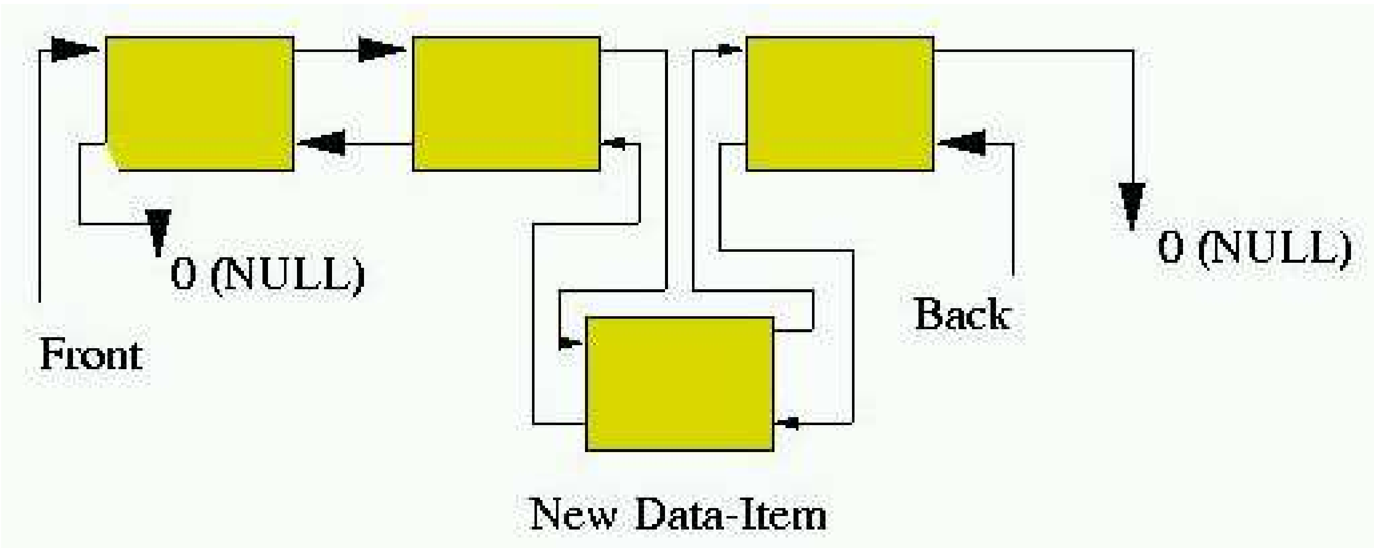

Figure 12.2: Adding a new element to a list

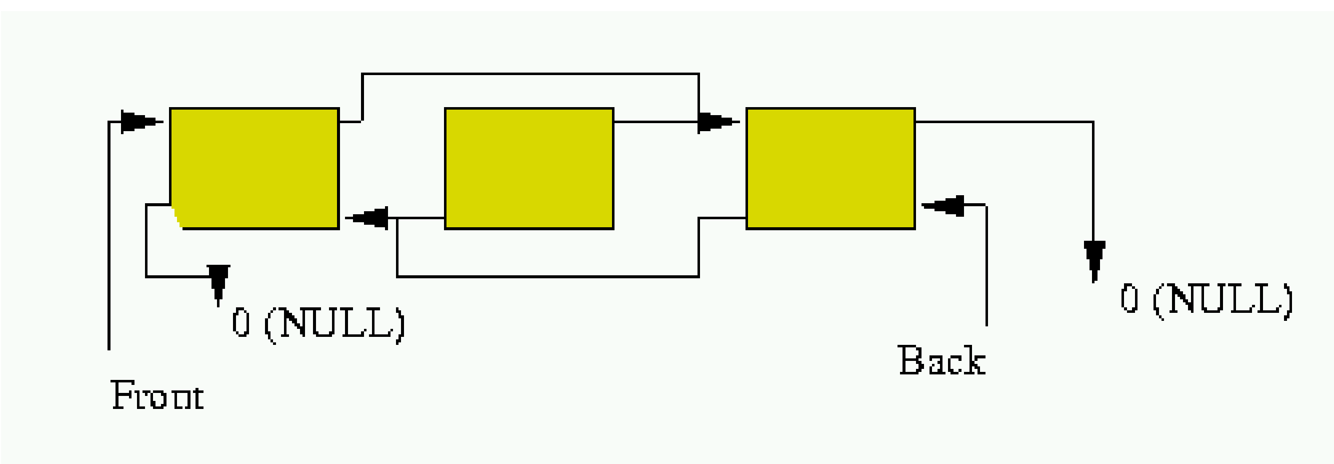

Figure 12.3: Removing an element from a list

As with the `vector`, it is an error to refer to an element of an empty list.

- A list may be initialized to a certain number of elements. By default, if the initialization value is not explicitly mentioned, the default value or default constructor for the actual data type is used. For example:

```
list<string> object(5, string("Hello")); // initialize to 5 Hello's
list<string> container(10);              // and to 10 empty strings
```

- A list may be initialized using a two iterators. To initialize a list with elements 5 until 10 (including the last one) of a `vector<string>` the following construction may be used:

```
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 `lists`, 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 to use this member only if the list is not empty.

  - `list::iterator list::begin()`:

    this member returns an iterator pointing to the first element in the list, returning `list::end()` if the list is empty.

  - `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 to use this member only if the list is not 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>
using namespace std;

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;
    list<string> second;

    first.push_back(string("alpha"));
    first.push_back(string("bravo"));
    first.push_back(string("golf"));
    first.push_back(string("quebec"));

    second.push_back(string("oscar"));
    second.push_back(string("mike"));
    second.push_back(string("november"));
    second.push_back(string("zulu"));

    first.merge(second);
    showlist(first);
}
```

  A subtlety is that `merge()` doesn't alter the list if the list itself is used as argument: `object.merge(object)` won't change the list '`object`'.

- `void 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 `value` is not provided, the extra elements are initialized to the default value of the used data type, otherwise `value` is used to initialize extra elements.
- `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>
using namespace std;

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.
- `size_t 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 therefore be available.

- `void list::splice(pos, object)`:

  this member function transfers the contents of `object` to the current list, starting the insertion at the iterator position `pos` of the object using the `splice()` member. Following `splice()`, `object` is empty. For example:

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

int main()
{
    list<string> object;

    object.push_front(string("Hello"));
    object.push_back(string("World"));

    list<string> argument(object);

    object.splice(++object.begin(), argument);

    cout << "Object contains " << object.size() << " elements, " <<
            "Argument contains " << argument.size() <<
            " elements," << endl;

    while (object.size())
    {
        cout << object.front() << endl;
        object.pop_front();
    }
}
```

  Alternatively, `argument` may be followed by a iterator of `argument`, indicating the first element of `argument` that should be spliced, or by two iterators `begin` and `end` defining the iterator-range `[begin, end)` on `argument` that should be spliced into `object`.

- `void list::swap()`:

  this member can be used to swap two lists using identical data types.

- `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 removing all multiply occurring words from the list:

```
    #include <iostream>
    #include <string>
```

```
#include <list>
using namespace std;
                        // see the merge() example
void showlist(list<string> &target);
void showlist(list<string> &target)
{
    for
    (
        list<string>::iterator from = target.begin();
        from != target.end();
        ++from
    )
        cout << *from << " ";

    cout << endl;
}


int main()
{
    string
        array[] =
        {
            "charley",
            "alpha",
            "bravo",
            "alpha"
        };

    list<string>
        target
        (
            array, array + sizeof(array)
            / sizeof(string)
        );

    cout << "Initially we have: " << endl;
    showlist(target);

    target.sort();
    cout << "After sort() we have: " << endl;
    showlist(target);

    target.unique();
    cout << "After unique() we have: " << endl;
    showlist(target);
}
/*
    Generated output:

    Initially we have:
    charley alpha bravo alpha
    After sort() we have:
    alpha alpha bravo charley
```

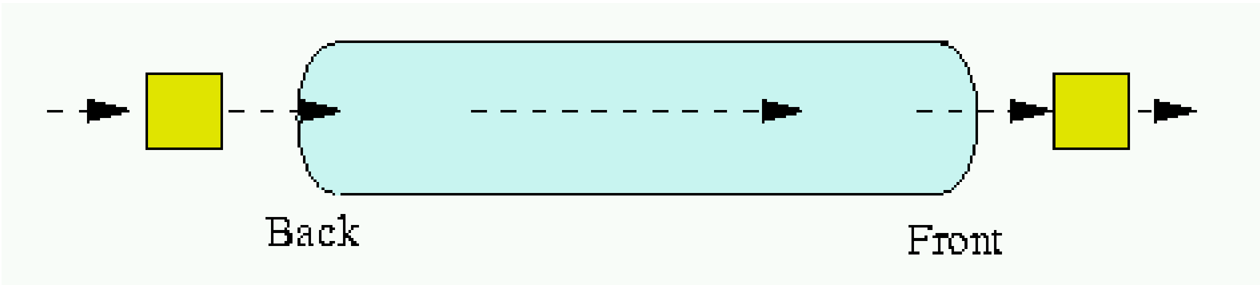

Figure 12.4: A queue data-structure

```
    After unique() we have:
    alpha bravo charley
*/
```

12.3.3 The 'queue' container

The `queue` class implements a queue data structure. Before `queue` containers can be used the following preprocessor directive must have been specified:

```
#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);
    ```

- The `queue` container only supports the basic operators for containers.
- 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 to use the member only if the queue is not 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 to use the member only if the queue is not 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. Nothing happens if the member is called for an empty queue. One might wonder why `pop()` returns `void`, instead of a value of type `Type` (cf. `front()`). Because of this, we must use `front()` first, and thereafter `pop()` to examine and remove the queue's front element. However, there is a good reason for this design. If `pop()` would return the container's front element, it would have to return that element by *value* rather than by *reference*, as a return by reference would create a dangling pointer, since `pop()` would also remove that front element. Return by *value*, however, is inefficient in this case: it involves at least one copy constructor call. Since it is impossible for `pop()` to return a value correctly and efficiently, it is more sensible to have `pop()` return no value at all and to require clients to use `front()` to inspect the value at the queue's front.

– `size_t 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. A queue can be emptied by:

- repeatedly removing its front element;
- assigning an empty queue using the same data type to it;
- having its destructor called.

12.3.4 The 'priority_queue' container

The `priority_queue` class implements a priority queue data structure. Before `priority_queue` containers can be used the following preprocessor directive must have been specified:

```
#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 ueue 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>
using namespace std;

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
{
    std::string d_s;

    public:
        Text(std::string const &str)
        :
            d_s(str)
        {}
        operator std::string const &() const
        {
            return d_s;
        }
        bool operator<(Text const &right) const
        {
            return d_s > right.d_s;
        }
};

using namespace std;

int main()
{
    priority_queue<Text> q;
    string word;
```

```
    while (cin >> word)
        q.push(word);

    while (q.size())
    {
        word = q.top();
        cout << word << 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. Nothing happens if this member is called for and empty priority queue. See section 12.3.3 for a discussion about the reason why `pop()` has return type `void`.
  - `size_t 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 to use the member only if the priority queue is not empty.

Note that the priority queue does not support iterators or a subscript operator. The only element that can be accessed is its top element. A priority queue can be emptied by:

- repeatedly removing its top element;
- assigning an empty queue using the same data type to it;
- having its destructor called.

12.3.5 The ‘deque’ container

The `deque` (pronounce: ‘deck’) class implements a doubly ended queue data structure (deque). Before `deque` containers can be used the following preprocessor directive must have been specified:

```
#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 available member functions. A `deque` is a combination of a `vector` and two queues, operating at both ends of the vector. In situations where random insertions and the addition and/or removal of elements at one or both sides of the vector occurs frequently, using a `deque` should be considered.

The following constructors, operators, and member functions are available for deques:

- Constructors:
  - 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 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
    deque<string> 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 to use the member only if the deque is not empty.
  - `deque::iterator deque::begin()`:

    this member returns an iterator pointing to the first element in the deque.
  - `void 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 to use the member only if the deque is not 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` starting at iterator 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 `value` is not provided, the additional elements are initialized to the default value of the used data type, otherwise `value` is used to initialize extra elements.

- `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.

- `size_t deque::size()`:

  this member returns the number of elements in the deque.

- `void deque::swap(argument)`:

  this member can be used to swap two deques using identical data types.

12.3.6 The 'map' container

The `map` class implements a (sorted) associative array. Before `map` containers can be used, the following preprocessor directive must have been specified:

```
#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 angle bracket notation, comparable to the specification we've seen with the `pair` (section 12.2) container. 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 to access its associated information. That information is called the *value*. For example, a phone book uses the names of people as the key, and uses the telephone number and maybe other information (e.g., the zip-code, the address, the profession) as the value. Since a `map` sorts its keys, the `key`'s `operator<()` must be defined, and it must be sensible to use it. For example, it is generally a bad idea to use pointers for keys, as sorting pointers is something different than sorting the values these pointers point to.

The two fundamental operations on maps are the storage of *Key/Value* combinations, and the retrieval of values, given their keys. The index operator, using a key as the index, can be used for both. If the index operator is used as *lvalue*, insertion will be performed. If it is used as *rvalue*, the key's associated value is retrieved. Each key can be stored only once in a `map`. If the same key is entered again, the new value replaces the formerly stored value, which is lost.

A specific key/value combination can 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 now be constructed using:

```
StringIntValue("Hello", 1);
```

Finally, `pairs` may be used to represent key/value combinations used by maps:

```
pair<string, int>("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 angle brackets `>`: this is obligatory, as the immediate concatenation of the two angle 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 two iterators. The iterators may either point to `value_type` values for the map to be constructed, or to plain `pair` objects (see section 12.2). If pairs are used, their `first` elements represent the keys, and their `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 written instead of `pair<string, int>` as well.

    When `begin` is the first iterator used to construct a map and `end` the second iterator, `[begin, end)` will be used to initialize the map. 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.

The map receives its own copies of the data to which the iterators point. This is illustrated by the following example:

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

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 is destroyed. At that point the array pairs has been constructed. Thereupon, the map constructs a temporary pair object, which is used to construct the map 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 type of the right-hand side of the assignment operator must be equal to (or promotable to) the type of the map's value part. E.g., to add or change the value of element `"two"` in a map, the following statement can be used:

```
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.

  – `size_t 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 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 the iterator `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 <map>
using namespace std;

int main()
{
    map<string, int> object;

    object["one"] = 1;

    map<string, int>::iterator it = object.find("one");

    cout << "'one' " <<
            (it == object.end() ? "not " : "") << "found\n";

    it = object.find("three");

    cout << "'three' " <<
            (it == object.end() ? "not " : "") << "found\n";
}
/*
    Generated output:
'one' found
'three' not found
*/
```

- `... map::insert()`:

  this member can be used to insert elements into the map. It will, however, not replace the values associated with already existing keys by new values. Its return value depends on the version of `insert()` that is called:

  * `pair<map::iterator, bool> insert(keyvalue)` inserts 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 having the `key` that was specified in `keyvalue`. The use of this variant of `insert()` is illustrated by the following example:

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main()
{
    pair<string, int> pa[] =
    {
        pair<string,int>("one", 10),
        pair<string,int>("two", 20),
        pair<string,int>("three", 30),
```

```
        };
        map<string, int> object(&pa[0], &pa[3]);

                // {four, 40} and ‘true’ is returned
        pair<map<string, int>::iterator, bool>
            ret = object.insert
                    (
                        map<string, int>::value_type
                        ("four", 40)
                    );

        cout << boolalpha;

        cout << ret.first->first << " " <<
            ret.first->second << " " <<
            ret.second << " " << object["four"] << endl;

                // {four, 40} and ‘false’ is returned
        ret = object.insert
                    (
                        map<string, int>::value_type
                        ("four", 0)
                    );

        cout << ret.first->first << " " <<
            ret.first->second << " " <<
            ret.second << " " << object["four"] << endl;
    }
    /*
        Generated output:

        four 40 true 40
        four 40 false 40
    */
```

Note the somewhat peculiar constructions like

```
cout << ret.first->first << " " << ret.first->second << ...
```

Realize that ‘`ret`’ is equal to the `pair` returned by the `insert()` member function. Its ‘`first`’ field is an iterator into the `map<string, int>`, so it can be considered a pointer to a `map<string, int>::value_type`. These value types themselves are pairs too, having ‘`first`’ and ‘`second`’ fields. Consequently, ‘`ret.first->first`’ is the *key* of the map value (a `string`), and ‘`ret.first->second`’ is the *value* (an `int`).

∗ `map::iterator insert(pos, keyvalue)`. This way a `map::value_type` may also be inserted into the map. `pos` is ignored, and an iterator to the inserted element is returned.

∗ `void insert(first, beyond)` inserts the (`map::value_type`) elements pointed to by the iterator range `[first, beyond)`.

– `map::iterator map::lower_bound(key)`:

  this member returns an iterator pointing to the first `keyvalue` element of which the `key` is at least equal to the specified `key`. If no such element exists, the function returns `map::end()`.

– `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.

- `size_t map::size()`:

  this member returns the number of elements in the map.

- `void map::swap(argument)`:

  this member can be used to swap two maps, using identical key/value types.

- `map::iterator map::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 `map::end()`. The following example illustrates the member functions `equal_range()`, `lower_bound()` and `upper_bound()`:

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

int main()
{
    pair<string, int> pa[] =
    {
        pair<string,int>("one", 10),
        pair<string,int>("two", 20),
        pair<string,int>("three", 30),
    };
    map<string, int> object(&pa[0], &pa[3]);
    map<string, int>::iterator it;

    if ((it = object.lower_bound("tw")) != object.end())
        cout << "lower-bound 'tw' is available, it is: " <<
                it->first << endl;

    if (object.lower_bound("twoo") == object.end())
        cout << "lower-bound 'twoo' not available" << endl;

    cout << "lower-bound two: " <<
            object.lower_bound("two")->first <<
            " is available\n";

    if ((it = object.upper_bound("tw")) != object.end())
        cout << "upper-bound 'tw' is available, it is: " <<
                it->first << endl;

    if (object.upper_bound("twoo") == object.end())
        cout << "upper-bound 'twoo' 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 ‘tw’ is available, it is: two
                lower-bound ‘twoo’ not available
                lower-bound two: two is available
                upper-bound ‘tw’ is available, it is: two
                upper-bound ‘twoo’ 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 listing all keys and values in a map:

```
    #include <iostream>
    #include <iomanip>
    #include <map>

    using namespace std;

    int main()
    {
        pair<string, int>
            pa[] =
            {
                pair<string,int>("one", 10),
                pair<string,int>("two", 20),
                pair<string,int>("three", 30),
            };
        map<string, int>
            object(&pa[0], &pa[3]);

        for
        (
            map<string, int>::iterator it = object.begin();
                it != object.end();
                    ++it
        )
            cout << setw(5) << it->first.c_str() <<
```

```
                setw(5) << it->second << endl;
}
/*
    Generated output:
  one   10
three   30
  two   20
*/
```

12.3.7 The 'multimap' container

Like the map, the multimap class implements a (sorted) associative array. Before multimap containers can be used the following preprocessor directive must have been specified:

```
#include <map>
```

The main difference between the map and the multimap is that the multimap supports multiple values associated with the same key, whereas the map contains single-valued keys. Note that the multimap also accepts multiple identical values associated with identical keys.

The map and the multimap have the same set of 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 12.3.6 for an overview of the multimap member functions. Some member functions, however, deserve additional attention when used in the context of the multimap container. These members are discussed below.

- size_t map::count(key):

  this member returns the number of entries in the multimap associated with the given key.

- ... multimap::erase():

  this member can be used to erase elements from the map:

  - size_t erase(key) erases all elements having the given key. The number of erased elements is returned.
  - void erase(pos) erases the single element pointed to by pos. Other elements possibly having the same keys are not erased.
  - void erase(first, beyond) erases all elements indicated by the iterator range [first, beyond).

- pair<multimap::iterator, multimap::iterator> 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::iterator 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.

Although the functions `lower_bound()` and `upper_bound()` act identically in the `map` and `multimap` containers, their operation in a `multimap` deserves some additional attention. The next example illustrates `multimap::lower_bound()`, `multimap::upper_bound()` and `multimap::equal_range` applied to a `multimap`:

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

int main()
{
    pair<string, int> pa[] =
    {
        pair<string,int>("alpha", 1),
        pair<string,int>("bravo", 2),
        pair<string,int>("charley", 3),
        pair<string,int>("bravo", 6),   // unordered ‘bravo’ values
        pair<string,int>("delta", 5),
        pair<string,int>("bravo", 4),
    };
    multimap<string, int> object(&pa[0], &pa[6]);

    typedef multimap<string, int>::iterator msiIterator;

    msiIterator it = object.lower_bound("brava");

    cout << "Lower bound for ‘brava’: " <<
            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");

    cout << "Equal range for ‘bravo’:\n";
    for (it = itPair.first; it != itPair.second; ++it)
        cout << it->first << ", " << it->second << endl;
    cout << "Upper bound: " << it->first << ", " << it->second << endl;
```

```
        cout << "Equal range for ‘brav’:\n";
        itPair = object.equal_range("brav");
        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
        Equal range for ‘bravo’:
        bravo, 2
        bravo, 6
        bravo, 4
        Upper bound: charley, 3
        Equal range for ‘brav’:
        Upper bound: bravo, 2
    */
```

In particular note the following characteristics:

- `lower_bound()` and `upper_bound()` produce the same result for non-existing keys: they both return the first element having a key that exceeds the provided key.
- Although the keys are ordered in the `multimap`, the values for equal keys are not ordered: they are retrieved in the order in which they were enterd.

12.3.8 The ‘set’ container

The `set` class implements a sorted collection of values. Before `set` containers can be used the following preprocessor directive must have been specified:

```
#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 value to be inserted into a `set` can be explicitly created: Every `set` defines a `value_type` which may be used to create values that can be stored in the `set`. For example, a value for a `set<string>` can be constructed as follows:

```
set<string>::value_type setValue("Hello");
```

The `value_type` is associated with the `set<string>`. Anonymous `value_type` objects are also often used. E.g.,

```
set<string>::value_type("Hello");
```

Instead of using the line `set<string>::value_type(...)` over and over again, a `typedef` is often used to reduce typing and to improve legibility:

```
typedef set<string>::value_type StringSetValue
```

Using this typedef, values for the `set<string>` may be constructed as follows:

```
StringSetValue("Hello");
```

Alternatively, values of the set's type may be used immediately. In that case the value of type `Type` is implicitly converted to a `set<Type>::value_type`.

The following constructors, operators, and member functions are available for the `set` container:

- Constructors:
  - A `set` may be constructed empty:
    ```
    set<int> object;
    ```
  - A set may be initialized using two iterators. For example:
    ```
    int intarr[] = {1, 2, 3, 4, 5};

    set<int> object(&intarr[0], &intarr[5]);
    ```

  Note that all values in the set must be different: it is not possible to store the same value repeatedly when the set is constructed. If the same value occurs repeatedly, only the first instance of the value 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. If the set is empty `set::end()` is returned.
  - `set::clear()`:

    this member erases all elements from the set.
  - `size_t 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 the iterator `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)` inserts a new `set::value_type` into the set. The return value is a `pair<set::iterator, bool>`. If the returned `bool` field is `true`, `value` was inserted into the set. The value `false` indicates that the value that was specified was already available in the set, and so the provided `value` was not inserted into the set. In both cases the `set::iterator` field points to the data element in the `set` having the specified `value`.
    * `set::iterator insert(pos, keyvalue)`. This way a `set::value_type` may also be 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 first `keyvalue` element of which the `key` is at least equal to the specified `key`. If no such element exists, the function returns `set::end()`.

- `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.

- `size_t set::size()`:

  this member returns the number of elements in the set.

- `void set::swap(argument)`:

  this member can be used to swap two sets (`argument` being the second set) that use identical data types.

- `set::iterator set::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 `set::end()`.

12.3.9 The 'multiset' container

Like the `set`, the `multiset` class implements a sorted collection of values. Before `multiset` containers can be used the following preprocessor directive must have been specified:

```
#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 12.3.8 for an overview of the `multiset` member functions. Some member functions, however, deserve additional attention when used in the context of the `multiset` container. These members are discussed below.

- `size_t set::count(value)`:

  this member returns the number of entries in the multiset associated with the given `value`.

- `... multiset::erase()`:

  this member can be used to erase elements from the set:

  - `size_t erase(value)` erases all elements having the given `value`. The number of erased elements is returned.
  - `void erase(pos)` erases the element pointed to by the iterator `pos`. Other elements possibly having the same values are not erased.
  - `void erase(first, beyond)` erases all elements indicated by the iterator range `[first, beyond)`.

- `pair<multiset::iterator, multiset::iterator> 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.

Although the functions `lower_bound()` and `upper_bound()` act identically in the `set` and `multiset` containers, their operation in a `multiset` deserves some additional attention. In particular note that with the `multiset` container `lower_bound()` and `upper_bound()` produce the same result for non-existing keys: they both return the first element having a key exceeding the provided key.

Here is an example showing the use of various member functions of a multiset:

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
```

```
string
    sa[] =
    {
        "alpha",
        "echo",
        "hotel",
        "mike",
        "romeo"
    };

multiset<string>
    object(&sa[0], &sa[5]);

object.insert("echo");
object.insert("echo");

multiset<string>::iterator
    it = object.find("echo");

for (; it != object.end(); ++it)
    cout << *it << " ";
cout << endl;

cout << "Multiset::equal_range(\"ech\")\n";
pair
<
    multiset<string>::iterator,
    multiset<string>::iterator
>
    itpair = object.equal_range("ech");

if (itpair.first != object.end())
    cout << "lower_bound() points at " << *itpair.first << endl;
for (; itpair.first != itpair.second; ++itpair.first)
    cout << *itpair.first << " ";

cout << endl <<
        object.count("ech") << " occurrences of 'ech'" << endl;

cout << "Multiset::equal_range(\"echo\")\n";
itpair = object.equal_range("echo");

for (; itpair.first != itpair.second; ++itpair.first)
    cout << *itpair.first << " ";

cout << endl <<
        object.count("echo") << " occurrences of 'echo'" << endl;

cout << "Multiset::equal_range(\"echoo\")\n";
itpair = object.equal_range("echoo");

for (; itpair.first != itpair.second; ++itpair.first)
    cout << *itpair.first << " ";
```

```
        cout << endl <<
                object.count("echoo") << " occurrences of 'echoo'" << endl;
    }
    /*
        Generated output:

        echo echo echo hotel mike romeo
        Multiset::equal_range("ech")
        lower_bound() points at echo

        0 occurrences of 'ech'
        Multiset::equal_range("echo")
        echo echo echo
        3 occurrences of 'echo'
        Multiset::equal_range("echoo")

        0 occurrences of 'echoo'
    */
```

12.3.10 The 'stack' container

The `stack` class implements a stack data structure. Before `stack` containers can be used the following preprocessor directive must have been specified:

```
#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: the lifetime of these variables is determined by the time these functions are active, contrary to global (or static local) variables, which live for as long as the program itself lives. Another example is found in calculators using the *Reverse Polish Notation* (RPN), in which the operands of operators are entered in the stack, whereas operators pop their operands off the stack and push the results of their work back onto the stack.

As an example of the use of a stack, consider figure 12.5, in which the contents of the stack is shown while the expression `(3 + 4) * 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.

From figure 12.5 we see that a stack has one point (the *top*) where items can be pushed onto and popped off the stack. This top element is the stack's only immediately visible element. It may be accessed and modified directly.

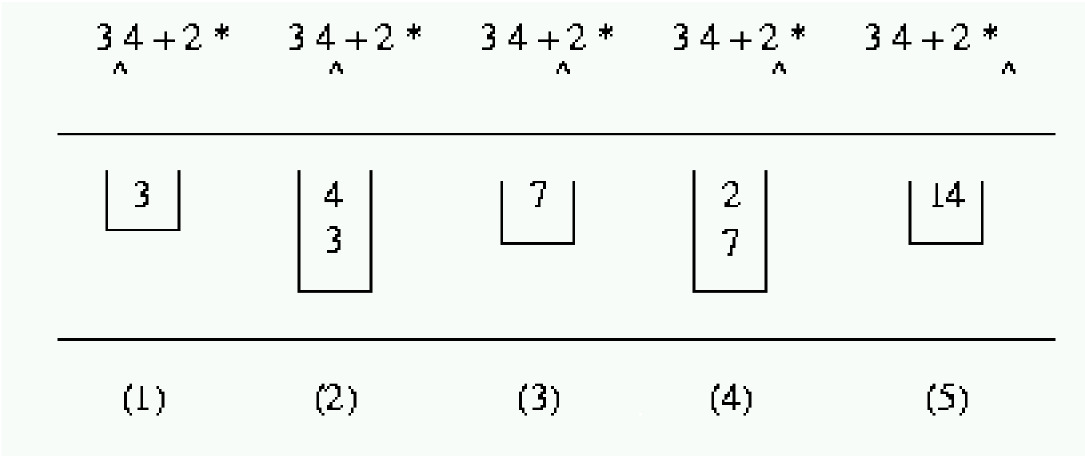

Figure 12.5: The contents of a stack while evaluating `3 4 + 2 *`

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);
    ```

- Only the basic set of container operators are supported by the `stack`
- 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. Nothing happens if `pop()` is used with an empty stack. See section 12.3.3 for a discussion about the reason why `pop()` has return type `void`.
  - `size_t stack::size()`:

    this member returns the number of elements in the stack.
  - `Type &stack::top()`:

    this member returns a reference to the stack's top (and only visible) element. It is the responsibility of the programmer to use this member only if the stack is not empty.

Note that the stack does not support iterators or a subscript operator. The only elements that can be accessed is its top element. A stack can be emptied by:

- repeatedly removing its front element;
- assigning an empty stack using the same data type to it;
- having its destructor called.

12.3.11 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. 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 9.10). 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. Before `hash_map` containers can be used the following preprocessor directive must have been specified:

```
#include <ext/hash_map>
```

The `hash_(multi)map` is not yet part of the ANSI/ISO standard. Once this container becomes part of the standard, it is likely that the `ext/` prefix in the `#include` preprocessor directive can be removed. Note that starting with the Gnu `g++` compiler version 3.2 the `__gnu_cxx` namespace is used for symbols defined in the `ext/` header files. See also section 2.1.

Constructors, operators and member functions available for the map are also available for the hash_map. The map and hash_map support the same set of operators and member functions. 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 additional constructor:

```
        hash_map<...> hash(n);
```

where n is a size_t 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 key 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 installed in the **C++** compiler's INCLUDE path as the file hashclasses.h, sources may specify the following preprocessor directive to make a set of classes available that can be used to instantiate a hash table

```
    #include <hashclasses.h>
```

Otherwise, sources must specify the following preprocessor directive:

```
    #include <ext/hash_map>
```

```
#ifndef _INCLUDED_HASHCLASSES_H_
#define _INCLUDED_HASHCLASSES_H_

#include <string>
#include <cctype>

/*
    Note that with the Gnu g++ compiler 3.2 (and beyond?) the ext/ header
    uses the __gnu_cxx namespace for symbols defined in these header files.

    When using compilers before version 3.2, do:
        #define __gnu_cxx   std
    before including this file to circumvent problems that may occur
    because of these namespace conventions which were not yet used in versions
    before 3.2.

*/

#include <ext/hash_map>
#include <algorithm>

/*
    This file is copyright (c) GPL, 2001-2004
    ==========================================
    august 2004: redundant include guards removed

    october 2002:   provisions for using the hashclasses with the g++ 3.2
                compiler were incorporated.
```

```
april 2002: namespace FBB introduced
            abbreviated class templates defined,
            see the END of this comment section for examples of how
            to use these abbreviations.

jan 2002:   redundant include guards added,
            required header files adapted,
            for_each() rather than transform() used

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, using int as the value type. Any other type can be used instead
          for the value type:

                                // key is char const *, case sensitive
    __gnu_cxx::hash_map<char const *, int, FBB::HashCharPtr,
                        FBB::EqualCharPtr >
        hashtab;

                                // key is char const *, case insensitive
    __gnu_cxx::hash_map<char const *, int, FBB::HashCaseCharPtr,
                                     FBB::EqualCaseCharPtr >
        hashtab;

                                // key is std::string, case sensitive
    __gnu_cxx::hash_map<std::string, int, FBB::HashString>
        hashtab;

                                // key is std::string, case insensitive
    __gnu_cxx::hash_map<std::string, int, FBB::HashCaseString,
                                    FBB::EqualCaseString>
        hashtab;
```

```
    Instead of the above full typedeclarations, the following shortcuts should
    work as well:

        FBB::CharPtrHash<int>       // key is char const *, case sensitive
            hashtab;

        FBB::CharCasePtrHash<int>   // key is char const *, case insensitive
            hashtab;

        FBB::StringHash<int>        // key is std::string, case sensitive
            hashtab;

        FBB::StringCaseHash<int>    // key is std::string, case insensitive
            hashtab;

    With these template types iterators and other map-members are also
    available. E.g.,

    --------------------------------------------------------------------------
    extern FBB::StringHash<int> dh;

    for (FBB::StringHash<int>::iterator it = dh.begin(); it != dh.end(); it++)
        std::cout << it->first << " - " << it->second << std::endl;
    --------------------------------------------------------------------------

    Feb. 2001 - April 2002
    Frank B. Brokken (f.b.brokken@rug.nl)
*/

namespace FBB
{

    class HashCharPtr
    {
        public:
            size_t operator()(char const *str) const
            {
                return __gnu_cxx::hash<char const *>()(str);
            }
    };

    class EqualCharPtr
    {
        public:
            bool operator()(char const *x, char const *y) const
            {
                return !strcmp(x, y);
            }
    };

    class HashCaseCharPtr
    {
        public:
            size_t operator()(char const *str) const
```

```
        {
            std::string s = str;
            for_each(s.begin(), s.end(), *this);
            return __gnu_cxx::hash<char const *>()(s.c_str());
        }
        void operator()(char &c) const
        {
            c = tolower(c);
        }
};

class EqualCaseCharPtr
{
    public:
        bool operator()(char const *x, char const *y) const
        {
            return !strcasecmp(x, y);
        }
};

class HashString
{
    public:
        size_t operator()(std::string const &str) const
        {
            return __gnu_cxx::hash<char const *>()(str.c_str());
        }
};

class HashCaseString: public HashCaseCharPtr
{
    public:
        size_t 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());
        }
};


template<typename Value>
class CharPtrHash: public
    __gnu_cxx::hash_map<char const *, Value, HashCharPtr, EqualCharPtr >
{
    public:
        CharPtrHash()
```

```
        {}
        template <typename InputIterator>
        CharPtrHash(InputIterator first, InputIterator beyond)
        :
            __gnu_cxx::hash_map<char const *, Value, HashCharPtr,
                                EqualCharPtr>(first, beyond)
        {}
};

template<typename Value>
class CharCasePtrHash: public
    __gnu_cxx::hash_map<char const *, Value, HashCaseCharPtr,
                                            EqualCaseCharPtr >
{
    public:
        CharCasePtrHash()
        {}
        template <typename InputIterator>
        CharCasePtrHash(InputIterator first, InputIterator beyond)
        :
            __gnu_cxx::hash_map<char const *, Value,
                        HashCaseCharPtr, EqualCaseCharPtr>
                        (first, beyond)
        {}
};

template<typename Value>
class StringHash: public __gnu_cxx::hash_map<std::string, Value,
                                            HashString>
{
    public:
        StringHash()
        {}
        template <typename InputIterator>
        StringHash(InputIterator first, InputIterator beyond)
        :
            __gnu_cxx::hash_map<std::string, Value, HashString>
                         (first, beyond)
        {}
};


template<typename Value>
class StringCaseHash: public
        __gnu_cxx::hash_map<std::string, int, HashCaseString,
                                EqualCaseString>
{
    public:
        StringCaseHash()
        {}
        template <typename InputIterator>
        StringCaseHash(InputIterator first, InputIterator beyond)
        :
            __gnu_cxx::hash_map<std::string,
```

```
                            int, HashCaseString,
                            EqualCaseString>(first, beyond)
            {}
    };

    template<typename Key, typename Value>
    class Hash: public
            __gnu_cxx::hash_map<Key, Value,
                        __gnu_cxx::hash<Key>(),
                        equal<Key>())
    {};

}
#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 <iostream>
        // the following header file must be available in the compiler’s
        // INCLUDE path:
    #include <hashclasses.h>
    using namespace std;
    using namespace FBB;

    int main()
    {
        __gnu_cxx::hash_map<string, int, HashString > months;
        // Alternatively, using the classes defined in hashclasses.h,
        // the following definitions could have been used:
        //      CharPtrHash<int> months;
        // or:
        //      StringHash<int> months;

        months["january"] = 31;
        months["february"] = 28;
        months["march"] = 31;
        months["april"] = 30;
        months["may"] = 31;
        months["june"] = 30;
        months["july"] = 31;
        months["august"] = 31;
        months["september"] = 30;
        months["october"] = 31;
        months["november"] = 30;
        months["december"] = 31;

        cout << "september -> " << months["september"] << 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.

Before the `hash_set` and `hash_multiset` containers can be used the following preprocessor directive must have been specified:

```
#include <ext/hash_set>
```

12.4 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.

Before `complex` containers can be used the following preprocessor directive must have been specified:

```
#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>

using namespace std;

int main()
{
    stack<complex<double> >
        cstack;

    cstack.push(complex<double>(3.14, 2.71));
    cstack.push(complex<double>(-3.14, -2.71));

    while (cstack.size())
    {
        cout << cstack.top().real() << ", " <<
                cstack.top().imag() << "i" << 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 primitve 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, accepting complex numbers as their arguments. For example,

```
abs(complex<double>(3, -5));
pow(target, complex<int>(2, 3));
```

- Complex numbers may be extracted from `istream` objects and inserted into `ostream` objects. The insertion results in an ordered pair `(x, y)`, in which `x` represents the real part and `y` the imaginary part of the complex number. The same form may also be used when extracting a complex number from an `istream` object. However, simpler forms are also allowed. E.g., `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**, programming problems are commonly approached using a top-down structured approach: functions and actions of the program are defined in terms of sub-functions, which again are defined in sub-sub-functions, etc.. This yields a hierarchy of code: `main()` at the top, followed by a level of functions which are called from `main()`, etc..

In **C++** the dependencies between code and data is also frequently defined in terms of dependencies among *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 described here is of a different kind: a class can be *defined* in terms of an older, pre-existing, class. This produces a new class having all the functionality of the older class, and additionally introducing its own specific functionality. Instead of composition, where a given class *contains* another class, we here refer to *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 discussing inheritance the existing class is called the *base class*, while the new class is called the *derived class*.

Derivation of classes is often used when the methodology of **C++** program development is fully exploited. In this chapter we will first address the syntactical possibilities offered by **C++** for deriving classes from other classes. Then we will address some of the resulting possibilities.

As we have seen in the introductory chapter (see section 2.4), in the object-oriented approach to problem solving classes are identified during the problem analysis, after which objects of the defined classes 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.

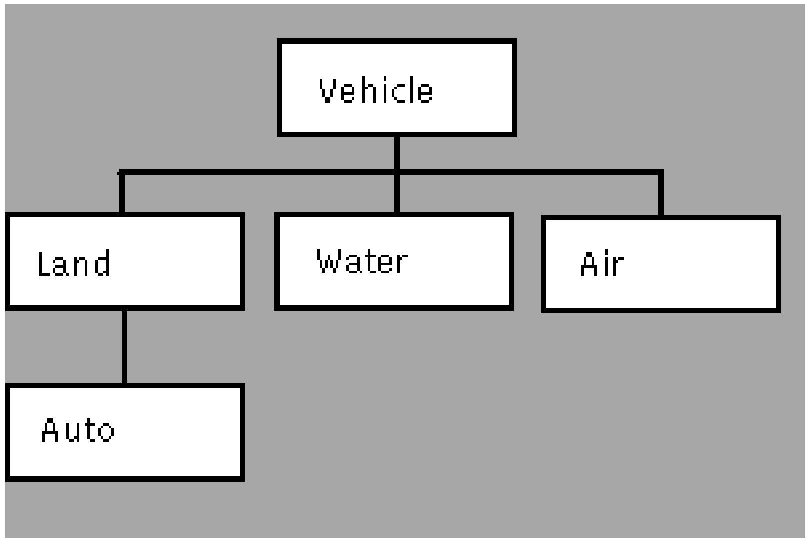

Figure 13.1: Initial object hierarchy of vehicles.

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: 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 in this class we implement the functionality to store and retrieve the vehicle's weight:

```
class Vehicle
{
    size_t d_weight;

    public:
        Vehicle();
        Vehicle(size_t weight);

        size_t weight() const;
        void setWeight(size_t weight);
};
```

Using this class, the vehicle's weight can be defined as soon as the corresponding object has been 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 `Vehicle`s and `Land`s could of course be represented using 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 needlessly bloat our code. E.g., consider the following code fragment which shows a class `Land` using composition (only the `setWeight()` functionality is shown):

```
class Land
{
    Vehicle d_v;        // composed Vehicle
    public:
        void setWeight(size_t weight);
};

void Land::setWeight(size_t weight)
{
    d_v.setWeight(weight);
}
```

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 achieved better by inheritance: `Land` is *derived* from `Vehicle`, in which `Vehicle` is the derivation's base class:

```
class Land: public Vehicle
{
    size_t d_speed;
    public:
        Land();
        Land(size_t weight, size_t speed);

        void setspeed(size_t speed);
        size_t speed() const;
};
```

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 of a constructor with two arguments and interface functions to access the `speed` data member. In the above example *public derivation* is used. **C++** also supports *private derivation* and *protected derivation*. In section 13.6 their differences are discussed. A simple example showing the possibilities of of the derived class `Land` is:

```
Land veh(1200, 145);

int main()
{
    cout << "Vehicle weighs " << veh.weight() << endl
         << "Speed is " << veh.speed() << endl;
}
```

This example shows two features of derivation. First, `weight()` is not mentioned as a member in `Land`'s interface. Nevertheless it is used in `veh.weight()`. 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: they can only be accessed by `Vehicle`'s own member functions. This means that `Land`'s member functions *must* use interface functions (like `weight()` and 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 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*, i.e., the data members, 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 new data member: a hidden pointer to a table of pointers to virtual functions. So, in this case recompilation is also necessary, as the class's data members have been silently modified. 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 conveniently derived from Land:

```
class Auto: public Land
{
    char *d_name;

    public:
        Auto();
        Auto(size_t weight, size_t speed, char const *name);
        Auto(Auto const &other);

        ~Auto();

        Auto &operator=(Auto const &other);

        char const *name() const;
        void setName(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 inheritance has 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 (size_t weight, size_t speed)
{
    setWeight(weight);
    setspeed(speed);
}
```

This implementation has the following disadvantage. The **C++** compiler will generate code calling the base class's default constructor from each constructor in the derived class, unless explicitly instructed otherwise. This can be compared to the situation we encountered 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 more efficient approach is of course to call the constructor of `Vehicle` expecting an `size_t weight` argument directly. The syntax achieving this is to mention the constructor to be called (supplied with its arguments) immediately following the argument list of the constructor of the derived class itself. Such a base class initializer is shown in the next example. Following the constructor's head a colon appears, which is then followed by the base class constructor. Only then any member initializer may be specified (using commas to separate multiple initializers), followed by the constructor's body:

```
Land::Land(size_t weight, size_t speed)
:
    Vehicle(weight)
{
    setspeed(speed);
}
```

13.3 The destructor of a derived class

Destructors of classes are automatically called when an object is destroyed. This also holds true for objects of classes derived from other classes. Assume we have the following situation:

```
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 exists. Hence, its destructor (`~Derived()`) is called. However, since `derived` is also a `Base` object, the `~Base()` destructor is called as well. It is *not* neccessary to call the base class destructor explicitly from the derived class destructor.

Constructors and destructors are called in a stack-like fashion: when `derived` is constructed, the appropriate base class constructor is called first, then the appropriate derived class constructor is called. When the object `derived` is destroyed, its destructor is called first, automatically followed by the activation of the `Base` class destructor. A derived class destructor is always called before its base class destructor is called.

13.4 Redefining member functions

The functionality of all members of a base class (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, pulling the second part, a trailer. Both the front engine and the trailer have their own weights, and the `weight()` 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 `size_t` 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
{
    size_t d_trailer_weight;

    public:
        Truck();
        Truck(size_t engine_wt, size_t speed, char const *name,
              size_t trailer_wt);

        void setWeight(size_t engine_wt, size_t trailer_wt);
        size_t weight() const;
};

Truck::Truck(size_t engine_wt, size_t speed, char const *name,
             size_t trailer_wt)
:
    Auto(engine_wt, speed, name)
{
```

```
        d_trailer_weight = trailer_wt;
    }
```

Note that the class `Truck` now contains two functions already present in the base class `Auto`: `setWeight()` and `weight()`.

- 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 `size_t` 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 `size_t` argument. To implement `Truck::setWeight()` we could write:

  ```
  void Truck::setWeight(size_t engine_wt, size_t trailer_wt)
  {
      d_trailer_weight = trailer_wt;
      Auto::setWeight(engine_wt);     // note: Auto:: is required
  }
  ```

- Outside of the class the `Auto`-version of `setWeight()` is accessed using 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 to include explicitly a member having the same function prototype as the base class member. This derived class member may then be implemented inline to call the base class member. This might be an elegant solution for the occasional situation. E.g., we add the following member to the `class Truck`:

  ```
  // in the interface:
  void setWeight(size_t engine_wt);

  // below the interface:
  inline void Truck::setWeight(size_t 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 additional function call is involved.

- The function `weight()` is also 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`:

  ```
  size_t Truck::weight() const
  {
      return
          (                           // sum of:
              Auto::weight() +     //   engine part plus
              d_trailer_weight        //   the trailer
          );
  }
  ```

The next example shows the actual use of the member functions of the class `Truck`, displaying several weights:

```
int main()
{
    Land veh(1200, 145);
    Truck lorry(3000, 120, "Juggernaut", 2500);

    lorry.Vehicle::setWeight(4000);

    cout << endl << "Truck weighs " <<
                    lorry.Vehicle::weight() << endl <<
        "Truck + trailer weighs " << lorry.weight() << endl <<
        "Speed is " << lorry.speed() << endl <<
        "Name is " << lorry.name() << endl;
}
```

Note the explicit call of `Vehicle::setWeight(4000)`: assuming `setWeight(size_t 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`.

With `Vehicle::weight()` and `Truck::weight()` the situation is somewhat different: here the function `Truck::weight()` is a *redefinition* of `Vehicle::weight()`, so in order to reach `Vehicle::weight()` a scope resolution operation (`Vehicle::`) is required.

13.5 Multiple inheritance

Up to now, a class was always derived from *a single* base class. **C++** also supports *multiple derivation*, in which a class is derived from several base classes and hence inherits 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. Otherwise, composition might be more appropriate. In general, linear derivation, in which there is only one base class, is used 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-operner, 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 allowing us to construct an instrument panel of an aircraft's cockpit. We design all kinds of instruments, like an artifical 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);

        size_t activeFrequency() const;
        size_t standByFrequency() const;
```

```
        void setStandByFrequency(size_t freq);
        size_t toggleActiveStandby();
        void setVolume(size_t level);
        void identEmphasis(bool on_off);
};
```

In the class'ss contructor we assume the availability of the classes `Intercom`, which is used by the pilot to listen to the information transmitted by 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);

        size_t frequency() const;
        size_t passiveFrequency() const;

        void setPassiveFrequency(size_t freq);
        size_t toggleFrequencies();

        void setAudioLevel(size_t level);
        void powerOn(bool on_off);
        void testState(bool on_off);
        void transmit(Message &message);
};
```

Using objects of this class we can receive messages, transmitted though the `Intercom`, but we can also *transmit* messages, using a `Message` object that's passed to the `ComSet` object using its `transmit()` member function.

Now we're ready to construct 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 often 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's code, but by the ordering of the base classes 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 both units in fact have a composed object Amplifier, handling the volume setting. A revised class might then either use a Amplifier &amplifier() 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 additional members will normally be defined as inline:

```
class NavComSet: public ComSet, public NavSet
{
    public:
        NavComSet(Intercom &intercom, VHF_Dial &dial);
        void comVolume(size_t volume);
        void navVolume(size_t volume);
};
inline void NavComSet::comVolume(size_t volume)
{
    ComSet::setVolume(volume);
```

```
}
inline void NavComSet::navVolume(size_t 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);
        void comVolume(size_t volume);
        void navVolume(size_t volume);
};
inline MyNavComSet::MyNavComSet(Intercom &intercom, VHF_Dial &dial)
:
    NavComSet(intercom, dial);
{}
inline void MyNavComSet::comVolume(size_t volume)
{
    ComSet::setVolume(volume);
}
inline void MyNavComSet::navVolume(size_t volume)
{
    NavSet::setVolume(volume);
}
```

13.6 Public, protected and private derivation

As we've seen, classes may be derived from other classes using inheritance. Usually the derivation type is `public`, implying that the access rights of the base class's interface is unaltered in the derived class.

Apart from public derivation, **C++** also supports *protected derivation* and *private derivation*

To use protected derivation. the keyword `protected` is specified in the inheritance list:

```
class Derived: protected Base
```

With protected derivation all the base class's public and protected members receive protected access rights in the derived class. Members having protected access rights are available to the class itself and to all classes that are (directly or indirectly) derived from it.

To use private derivation. the keyword `private` is specified in the inheritance list:

```
class Derived: private Base
```

With private derivation all the base class's members receive private access rights in the derived class. Members having private access rights are only available to the class itself.

Combinations of inheritance types do occur. For example, when designing a stream-class it is usually derived from `std::istream` or `std::ostream`. However, before a stream can be constructed, a `std::streambuf` must be available. Taking advantage of the fact that the inheritance order is taken seriously by the compiler, we can use multiple inheritance (see section 13.5) to derive the class from both `std::streambuf` and (then) from, e.g., `std::ostream`. As our class faces its clients as a `std::ostream` and not as a `std::streambuf`, we use private derivation for the latter, and public derivation for the former class:

```
class Derived: private std::streambuf, public std::ostream
```

13.7 Conversions between base classes and derived classes

When inheritance is used to define 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.

13.7.1 Conversions in object assignments

Continuing our discussion of the `NavCom` class, introduced in section 13.5 We start by defining two objects, a base class and a derived class object:

```
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`, allowing the assignment *from* `navcom` (a derived class object) *to* `com` (a base class object):

```
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. Although derived class objects are also base class objects, the reverse does not hold true: a base class object is not also a derived class object.

The following general rule applies: in assignments in which base class objects and derived class objects are involved, assignments in which data are dropped 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 responsibility of the programmere constructing the assignment operator to decide what to do with the missing data.

13.7.2 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 manipulating `weight` can be called as this is the `Vehicle`'s *only* functionality. As far as the compiler can tell this is the object `vp` points to.

The same reasoning holds true for references to `Vehicles`. If, e.g., a function is defined having a `Vehicle` reference parameter, the function may be passed an object of a class derived from `Vehicle`. Inside the function, the specific `Vehicle` members remain accessible. This analogy between pointers and references holds true in general. Remember that a reference is nothing but a pointer in disguise: it mimics a plain variable, but actually it is a pointer.

This restricted functionality furthermore has an important consequence for the class `Truck`. After the statement `vp = &truck`, `vp` points to a `Truck` object. So, `vp->weight()` will return 2600 instead of 8600 (the combined weight of the cabin and of the trailer: 2600 + 6000), which would have been returned by `truck.weight()`.

When a function is called using a pointer to an object, then the *type of the pointer* (and not the type of the object) determines which member functions are available and executed. In other words, **C++** implicitly converts the type of an object reached through a pointer to the pointer's type.

If the actual type of the object 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->name() << 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 also 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->weight()`. The reason for this should now be clear: `vp` calls `Vehicle::weight()` and not `Truck::weight()`, even though `vp` actually points to a `Truck`.

Fortunately, a remedy is available. In **C++** a `Vehicle *vp` may call a function `Truck::weight()` 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..[1]

Polymorphism is realized by a feature called *late binding*. It's called that way because the decision *which* function to call (a base class function or a function of a derived class) cannot be made *compile-time*, but is postponed until the program is actually executed: only then it is determined which member function will actually be called.

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

[1] 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?"

`weight()` and `setWeight()` might well be declared `virtual`. The relevant sections of the class definitions of the class `Vehicle` and `Truck` are shown below. Also, we show the implementations of the member functions `weight()` of the two classes:

```
class Vehicle
{
    public:
        virtual int weight() const;
        virtual void setWeight(int wt);
};

class Truck: public Vehicle
{
    public:
        void setWeight(int engine_wt, int trailer_wt);
        int weight() const;
};

int Vehicle::weight() const
{
    return (weight);
}

int Truck::weight() const
{
    return (Auto::weight() + trailer_wt);
}
```

Note that the keyword `virtual` *only* needs to appear in 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 `weight()` is declared `virtual` in `Auto`, rather than in `Vehicle`. The specific characteristics of virtual member functions would then, for the member function `weight()`, only appear with `Auto` (and its derived classes) pointers or references. With a `Vehicle` pointer, static binding would 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", 15000);     // make Scania, trailer weight 15000
Vehicle *vp;                // generic vehicle pointer

int main()
{
    vp = &v;                                // see (1) below
    cout << vp->weight() << endl;

    vp = &t;                                // see (2) below
    cout << vp->weight() << endl;

    cout << vp->speed() << endl;     // see (3) below
}
```

Since the function `weight()` is defined `virtual`, late binding is used:

- at (1), `Vehicle::weight()` is called.
- at (2) `Truck::weight()` is called.
- at (3) a syntax error is generated. The member `speed()` is no member of `Vehicle`, and hence not callable via a `Vehicle*`.

The example illustrates that when a pointer to a class is used *only the functions which are members of that class can be called*. These functions *may* be `virtual`. However, this only influences the type of binding (early vs. late) and not the set of member functions that is visible to the pointer.

A virtual member function cannot be a static member function: a virtual member function is still an 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 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 `Vehicle`'s destructor is called, instead of the `Land` object's destructor. 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's destructor from being completed (or, better: started). A Bad Thing.

In **C++** this problem is solved using *virtual destructors*. 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 size_t weight() const;
};
```

By declaring a virtual destructor, the above `delete` operation (`delete vp`) will correctly call `Land`'s destructor, rather than `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.

- This destructor should be defined as a *virtual* destructor if the class contains at least one virtual member function, to prevent incomplete destruction of derived class objects when destroying objects using base class pointers or references pointing to derived class objects (see the initial paragraphs of this section)

In the second case, the destructor doesn't have any special tasks to perform. In these cases the virtual destructor is given an empty body. For example, the definition of `Vehicle::~Vehicle()` may be as simple as:

```
Vehicle::~Vehicle()
{}
```

Often the destructor will be defined inline below the class interface.

temporary note: With the gnu compiler 4.1.2 an annoying bug prevents *virtual destructors* to be defined inline below their class interfaces without explicitly declaring the virtual destructor as inline within the interface. Until the bug has been repaired, inline virtual destructors should be defined as follows (using the class `Vehicle` as an example):

```
class Vehicle
{
    ...
    public:
        inline virtual ~Vehicle();  // note the 'inline'
        ...
};

inline Vehicle::~Vehicle()          // inline implementation
{}                                  // is kept unaltered.
```

14.3 Pure virtual functions

Until now the base class `Vehicle` contained its own, concrete, implementations of the virtual functions `weight()` and `setWeight()`. In **C++** it is 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, in some languages (like **C#, Delphi** and **Java**) known as an *interface*, defines a *protocol*, which *must* be implemented by 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* or *abstract base classes*. Abstract base classes are the foundation of many *design patterns* (cf. *Gamma et al.* (1995)) , allowing the programmer to create highly *reusable software*. Some of these design patterns are covered by the Annotations (e.g, the *Template Method* in section 20.3), but for a thorough discussion of Design Patterns the reader is referred to Gamma *et al.*'s book.

Functions that are only declared in the base class are called *pure virtual functions*. A function is made pure virtual by prefixing the keyword `virtual` to its declaration 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 std::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, than *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.

14.3.1 Implementing pure virtual functions

Pure virtual member functions may be implemented. To implement a pure virtual member function: pure virtual and implemented member function, provide it with its normal `= 0;` specification, but implement it nonetheless. Since the `= 0;` ends in a semicolon, the pure virtual member is always at most a declaration in its class, but an implementation may either be provided in-line below the class interface or it may be defined as a non-inline member function in a source file of its own.

Pure virtual member functions may be called from derived class objects or from its class or derived class members by specifying the base class and scope resolution operator with the function to be called. The following small program shows some examples:

```
#include <iostream>

class Base
{
    public:
        virtual ~Base();
        virtual void pure() = 0;
};
```

```
inline Base::~Base()
{}

inline void Base::pure()
{
    std::cout << "Base::pure() called\n";
}

class Derived: public Base
{
    public:
        virtual void pure();
};

inline void Derived::pure()
{
    Base::pure();
    std::cout << "Derived::pure() called\n";
}

int main()
{
    Derived derived;

    derived.pure();
    derived.Base::pure();

    Derived *dp = &derived;

    dp->pure();
    dp->Base::pure();
}
// Output:
//      Base::pure() called
//      Derived::pure() called
//      Base::pure() called
//      Base::pure() called
//      Derived::pure() called
//      Base::pure() called
```

Implementing a pure virtual function has limited use. One could argue that the pure virtual function's implementation may be used to perform tasks that can already be performed at the base-class level. However, there is no guarantee that the base class virtual function will actually be called from the derived class overridden version of the member function (like a base class constructor that is automatically called from a derived class constructor). Since the base class implementation will therefore at most be called optionally its functionality could as well be implemented in a separate member, which can then be called without the requirement to mention the base class explicitly.

14.4 Virtual functions in multiple inheritance

As mentioned in chapter 13 a class may be derived from multiple 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.

Consider what would happen if more than one 'path' would lead 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
{
    int d_field;
    public:
        void setfield(int val);
        int field() const;
};
inline void Base::setfield(int val)
{
    d_field = val;
}
inline int field() const
{
    return d_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, C++ compilers will normally refuse to generate code and will (correctly) identify an error.

The above code clearly duplicates its base class in the derivation, which can of course easily be avoided by not doubly deriving from `Base`. But duplication of a base class can also occur through nested inheritance, where an object is derived from, e.g., 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[2]. An `AirAuto` would ultimately contain two `Vehicles`, and hence two `weight` fields, two `setWeight()` functions and two `weight()` functions.

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

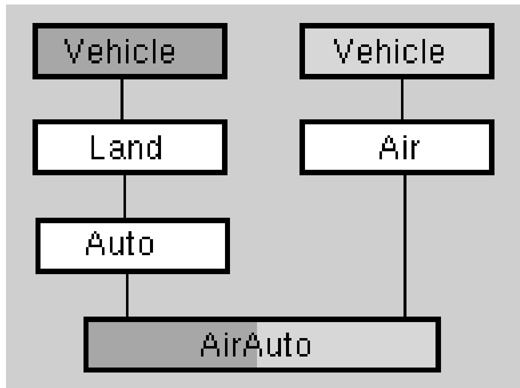

Figure 14.1: Duplication of a base class in multiple derivation.

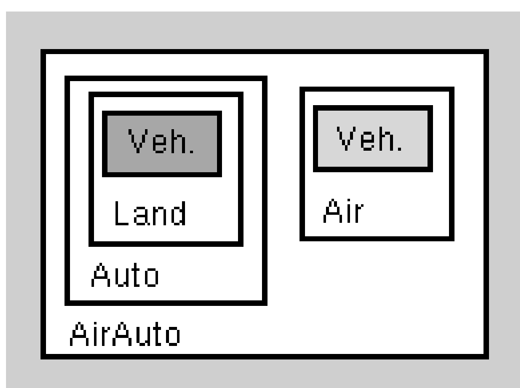

Figure 14.2: Internal organization of an `AirAuto` object.

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.weight() << endl;
```

The question of which member function weight() 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. The ambiguity is resolved using the scope resolution operator:

  ```
  // let's hope that the weight is kept in the Auto
  // part of the object..
  cout << cool.Auto::weight() << endl;
  ```

  Note the position of the scope operator and the class name: before the name of the member function itself.

- Second, a dedicated function weight() could be created for the class AirAuto:

  ```
  int AirAuto::weight() const
  {
      return Auto::weight();
  }
  ```

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, discussed in the next section.

14.4.2 Virtual base classes

As illustrated in Figure 14.2, an AirAuto represents *two* Vehicles. 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 an AirAuto is only one Vehicle, yet used multiple derivation. This is realized by defining the base class that is multiply mentioned in a derived class' inheritance tree 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 Auto: public Land
{
```

[2]such as the one in James Bond vs. the Man with the Golden Gun...

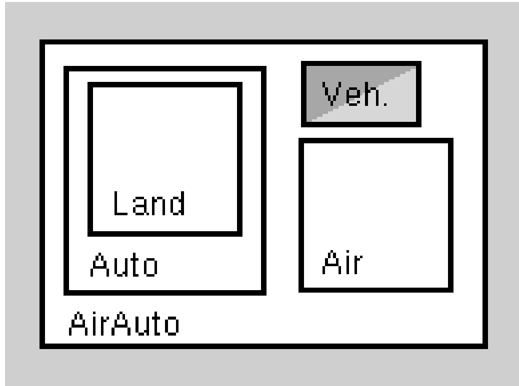

Figure 14.3: Internal organization of an `AirAuto` object when the base classes are virtual.

```
    // etc
};


class Air: virtual public Vehicle
{
    // etc
};

class AirAuto: public Auto, 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. Note the following:

- When base classes of a class using multiple derivation are themselves virtually derived from a base class (as shown above), the base class constructor normally called when the derived class constructor is called, is no longer used: 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 cases when constructing 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 when member functions of a base class are called. 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
{
    int d_trailer_weight;

    public:
        Truck();
        Truck(int engine_wt, int sp, char const *nm,
               int trailer_wt);

        void setWeight(int engine_wt, int trailer_wt);
        int weight() const;
};

Truck::Truck(int engine_wt, int sp, char const *nm,
              int trailer_wt)
:
    Auto(engine_wt, sp, nm)
{
    d_trailer_weight = trailer_wt;
}

int Truck::weight() const
{
    return                  // sum of:
        Auto::weight() +    //   engine part plus
        trailer_wt;         //   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 d_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);
};
```

```
inline 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 weight() const;
    };

inline Truck::Truck(int engine_wt, int sp, char const *nm,
                    int trailer_wt)
:
    Auto(engine_wt, sp, nm),
    TrailerVeh(trailer_wt)
{}

    inline int Truck::weight() const
    {
        return                      // sum of:
            Auto::weight() +        //   engine part plus
            TrailerVeh::weight();   //   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 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. Nonetheles, **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. This is called *down-casting*.
- 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 using 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();
    };
inline char const *Derived::toString()
{
    return "Derived object";
}

    int main()
    {
        Base *bp;
        Derived *dp,
        Derived d;

        bp = &d;

        dp = dynamic_cast<Derived *>(bp);

        if (dp)
            cout << dp->toString() << 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 only declared):

```
    class Base
    {
        public:
            virtual ~Base();
    };
    class Derived1: public Base;
    class Derived2: public Base;

    int main()
    {
        Base *bp;
        Derived1 *d1,
        Derived1 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>

    class Base
    {
        public:
            virtual ~Base();
            virtual char const *toString();
    };
    inline Base::~Base()
    {}
    inline char const *Base::toString()
    {
        return "Base::toString() called";
    }

    class Derived1: public Base
    {};

    class Derived2: public Base
    {};

    void process(Base &b)
    {
        try
        {
            std::cout << dynamic_cast<Derived1 &>(b).toString() << std::endl;
        }
        catch (std::bad_cast)
        {}

        try
        {
            std::cout << dynamic_cast<Derived2 &>(b).toString() << std::endl;
        }
        catch (std::bad_cast)
        {
            std::cout << "Bad cast to Derived2\n";
        }
    }

    int main()
    {
```

```
        Derived1 d;

        process(d);
    }
    /*
        Generated output:

        Base::toString() called
        Bad cast to Derived2
    */
```

In this example the value `std::bad_cast` is introduced. The `std::bad_cast` exception is thrown if the dynamic cast of a reference to a derived class object fails.

Note the form of the `catch` clause: `bad_cast` is the name of a type. In section 16.4.1 the construction of such a type is discussed.

The dynamic cast operator is a useful tool when an existing base class cannot or should not be modified (e.g., when the sources are not available), and a derived class may be modified instead. Code receiving a base class pointer or reference may then perform a dynamic cast to the derived class to access the derived class's functionality.

Casts from a base class reference or pointer to a derived class reference or pointer are called *downcasts*.

One may wonder what the difference is between a `dynamic_cast` and a `reinterpret_cast`. Of course, the `dynamic_cast` may be used with references and the `reinterpret_cast` can only be used for pointers. But what's the difference when both arguments are pointers?

When the `reinterpret_cast` is used, we tell the compiler that it literally should re-interpret a block of memory as something else. A well known example is obtaining the individual bytes of an `int`. An `int` consists of `sizeof(int)` bytes, and these bytes can be accessed by reinterpreting the location of the `int` value as a `char *`. When using a `reinterpret_cast` the compiler offers absolutely no safeguard. The compiler will happily `reinterpret_cast` an `int *` to a `double *`, but the resulting dereference produces at the very least a meaningless value.

The `dynamic_cast` will also reinterpret a block of memory as something else, but here a run-time safeguard is offered. The dynamic cast fails when the requested type doesn't match the actual type of the object we're pointing at. The `dynamic_cast`'s purpose is also much more restricted than the `reinterpret_cast`'s purpose, as it should only be used for downcasting to derived classes having virtual members.

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. Such `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 `type_id` 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 at least one virtual function
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 these static, built-in text-strings are required. Check out what your compiler produces in case of doubt.

In situations where the `typeid` operator is applied to determine the type of a derived class, it is important to realize that a base class *reference* should be 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
    ...
if (typeid(*bp) == typeid(Derived))     // 5: true
    ...
if (typeid(*bp) == typeid(Base))        // 6: false
    ...

Base &br = *bp;

if (typeid(br) == typeid(Derived))      // 7: true
    ...
if (typeid(br) == typeid(Base))         // 8: false
    ...
```

Here, `(1)` returns `false` as a `Base *` is not a `Derived *`. `(2)` returns `true`, as the two pointer types are the same, `(3)` and `(4)` return `false` as pointers to objects are not the objects themselves.

On the other hand, if `*bp` is used in the above expressions, then `(1)` and `(2)` return `false` as an object (or reference to an object) is not a pointer to an object, whereas `(5)` now returns `true`: `*bp` actually refers to a `Derived` class object, and `typeid(*bp)` will return `typeid(Derived)`. A similar result is obtained if a base class reference is used: `7` returning `true` and `8` returning `false`.

When a `0`-pointer is passed to the `operator typeid` a `bad_typeid` exception is thrown.

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 overriden 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 == endOfFile()` then the input device must be reset one character, otherwise `c` must be prepended to the characters to be read. The function will return `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 `u(nder)flow()` 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 also moves the pending character that is returned 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`, when calling this function and it returns `c` it may be assumed that the character `c` is 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` derived from `streambuf`. The problem is to construct a streambuf writing its information to the standard output stream in such a way that all white-space delimited 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
{
```

```
    int d_last;

    public:
        CapsBuf()
        :
            d_last(' ')
        {}

    protected:
        int overflow(int c)             // interface to the device.
        {
            std::cout.put(isspace(d_last) ? toupper(c) : c);
            return d_last = c;
        }
};
```

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;

        return 0;
    }
    /*
        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 realized by 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 `main()` function above:

```
    cin >> out.rdbuf();
```

We compile and link the program to the executable `caps`, and start:

```
echo hello world | caps
```

Unfortunately, nothing happens.... Nor do we get any reaction when 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 performs a `streambuf`-to-`streambuf` copy only 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 required `ios::out` mode by defining an output buffer using `setp()`. We do this by defining a buffer, but don't want to use it, so we let its size be 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 defines `CapsBuf` as an output buffer, thus activating the

```
istream::operator>>(streambuf *)
```

member. As the variable `dummy` is not used by `setp()` it may be defined as a local variable. It's only purpose in life it to indicate to `setp()` that no buffer is used. Here is the revised constructor of the class `CapsBuf`:

```
CapsBuf::CapsBuf()
:
    d_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.

By now it will 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 std::ostream &operator<<(std::ostream &str,
                                    Exception const &e);
    std::string d_reason;

    public:
        virtual ~Exception();
        virtual void process() const = 0;
        virtual operator std::string() const;
    protected:
        Exception(char const *reason);
};

    inline Exception::~Exception()
    {}
    inline Exception::operator std::string() const
    {
        return d_reason;
    }
    inline Exception::Exception(char const *reason)
    :
        d_reason(reason)
    {}
    inline std::ostream &operator<<(std::ostream &str, Exception const &e)
    {
        return str << e.operator std::string();
    }

#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);
        void process() const;
};
    inline FatalException::FatalException(char const *reason)
    :
        Exception(reason)
    {}
    inline void FatalException::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"
using namespace std;

void initialExceptionHandler(Exception const *e)
{
    cout << *e << endl;         // show the plain-text information

    if
    (
        !dynamic_cast<MessageException const *>(e)
        &&
        !dynamic_cast<WarningException const *>(e)
    )
        throw;                  // Pass on other types of Exceptions

    e->process();               // Process a message or a warning
    delete e;
}
```

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->weight()` 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 *vpointer*, 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 storing the addresses of the class's virtual functions.

Consequently, a statement like `vp->weight()` 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 `weight()` 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 figures Figure 14.4 and Figure 14.5 (provided by `Guillaume Caumon`[3]).

As can be seen from figures Figure 14.4 and Figure 14.5, 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 `weight()`: therefore, this class needs its own table of function pointers.

14.9 Undefined reference to vtable ...

Occasionaly, the linker will complain with a message like the following:

```
In function
    'Derived::Derived[in-charge]()':
    : undefined reference to 'vtable for Derived'
```

This error is caused by the absence of the implementation of a virtual function in a derived class, while the function is mentioned in the derived class's interface.

[3] mailto:Guillaume.Caumon@ensg.inpl-nancy.fr

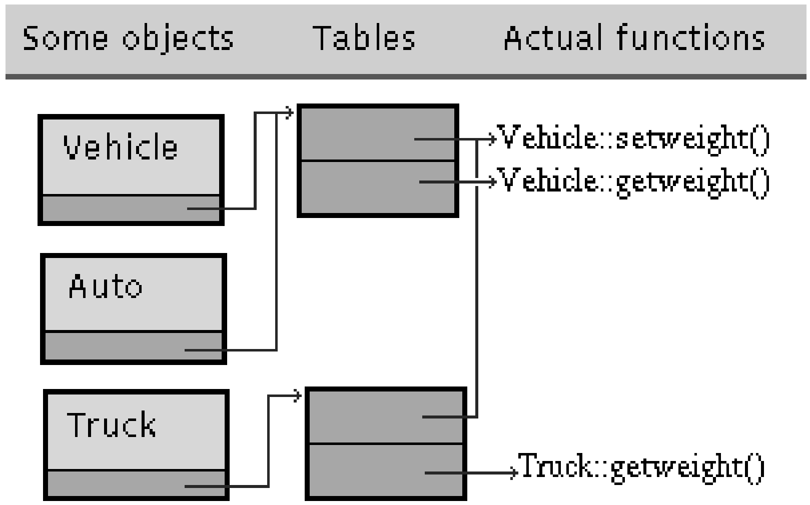

Figure 14.4: Internal organization objects when virtual functions are defined.

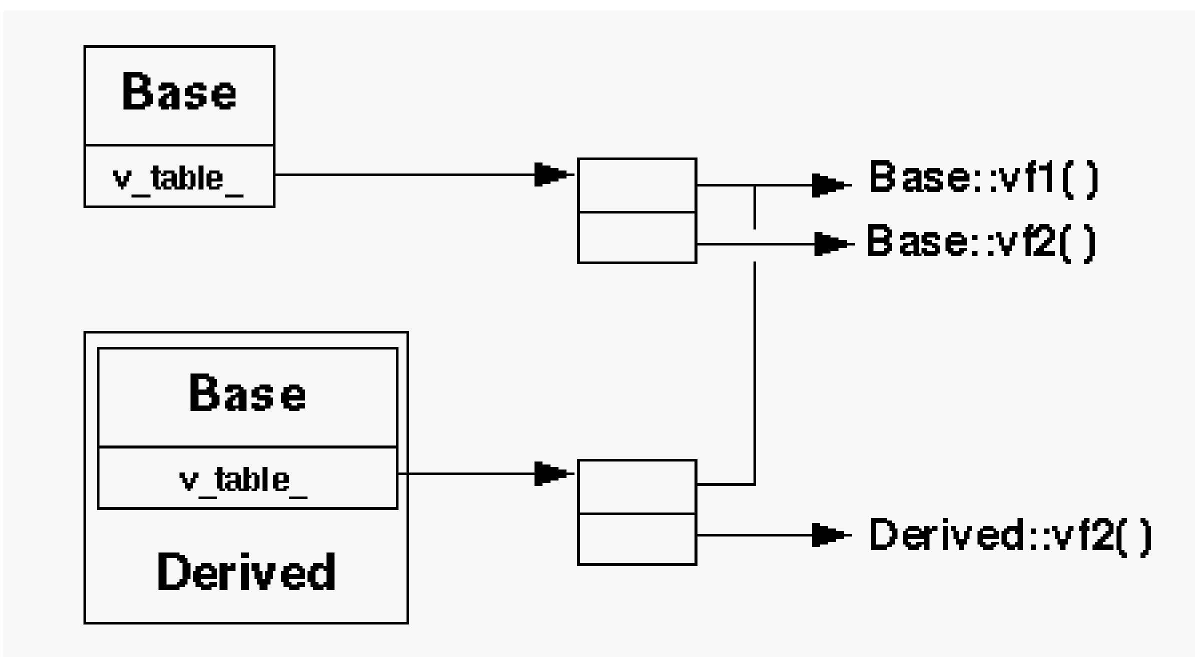

Figure 14.5: Complementary figure, provided by Guillaume Caumon

Such a situation can easily be created:

- Construct a (complete) base class defining a virtual member function;
- Construct a Derived class which mentions the virtual function in its interface;
- The Derived class's virtual function, overriding the base class's function having the same name, is not implemented. Of course, the compiler doesn't know that the derived class's function is not implemented and will, when asked, generate code to create a derived class object;
- However, the linker is unable to find the derived class's virtual member function. Therefore, it is unable to construct the derived class's vtable;
- The linker complains with the message:

```
undefined reference to 'vtable for Derived'
```

Here is an example producing the error:

```
class Base
{
    public:
        virtual void member();
};

    inline void Base::member()
    {}

class Derived
{
    public:
        virtual void member();      // only declared
};

int main()
{
    Derived d;  // Will compile, since all members were declared.
                // Linking will fail, since we don't have the
                // implementation of Derived::member()
}
```

It's of course easy to correct the error: implement the derived class's missing virtual member function.

14.10 Virtual constructors

As we have seen (section 14.2) **C++** supports *virtual destructors*. Like many other object oriented languages (e.g., **Java**), however, the notion of a *virtual constructor* is not supported. The absence of a virtual constructor turns into a problem when only a base class reference or pointer is available, and a copy of a derived class object is required. *Gamma et al.* (1995) developed the *Prototype Design Pattern* to deal with this situation.

In the Prototype Design Pattern each derived class is given the task to make available a member function returning a pointer to a new copy of the object for which the member is called. The usual name for this function is `clone()`. A base class supporting 'cloning' only needs to define a virtual destructor, and a *virtual copy constructor*, a pure virtual function, having the prototype `virtual Base *clone() const = 0`.

Since `clone()` is a pure virtual function all derived classes must implement their own 'virtual constructor'.

This setup suffices in most situations where we have a pointer or reference to a base class, but fails for example with abstract containers. We can't create a `vector<Base>`, with `Base` featuring the pure virtual `copy()` member in its interface, as `Base()` is called to initialize new elements of such a vector. This is impossible as `clone()` is a pure virtual function, so a `Base()` object can't be constructed.

The intuitive solution, providing `clone()` with a default implementation, defining it as an ordinary virtual function, fails too as the container calls the normal `Base(Base const &)` copy constructor, which would then have to call `clone()` to obtain a copy of the copy constructor's argument. At this point it becomes unclear what to do with that copy, as the new `Base` object already exists, and contains no `Base` pointer or reference data member to assign `clone()`'s return value to.

An alternative and preferred approach is to keep the original `Base` class (defined as an abstract base class), and to manage the `Base` pointers returned by `clone()` in a separate class `Clonable()`. In chapter 16 we'll encounter means to merge `Base` and `Clonable` into one class, but for now we'll define them as separate classes.

The class `Clonable` is a very standard class. As it contains a pointer member, it needs a copy constructor, destructor, and overloaded assignment operator (cf. chapter 7). It's given at least one non-standard member: `Base &get() const`, returning a reference to the derived object to which `Clonable`'s `Base *` data member refers, and optionally a `Clonable(Base const &)` constructor to allow promotions from objects of classes derived from `Base` to `Clonable`.

Any non-abstract class derived from `Base` must implement `Base *clone()`, returning a pointer to a newly created (allocated) copy of the object for which `clone()` is called.

Once we have defined a derived class (e.g., `Derived1`), we can put our `Clonable` and `Base` facilities to good use.

In the next example we see `main()` in which a `vector<Clonable>` was defined. An anonymous `Derived1` object is thereupon inserted into the vector. This proceeds as follows:

- The anonymous `Derived1` object is created;
- It is promoted to `Clonable`, using `Clonable(Base const &)`, calling `Derived1::clone()`;
- The just created `Clonable` object is inserted into the vector, using `Clonable(Clonable const &)`, again using `Derived1::clone()`.

In this sequence, two temporary objects are used: the anonymous object and the `Derived1` object constructed by the first `Derived1::clone()` call. The third `Derived1` object is inserted into the vector. Having inserted the object into the vector, the two temporary objects are destroyed.

Next, the `get()` member is used in combination with `typeid` to show the actual type of the `Base &` object: a `Derived1` object.

The most interesting part of main() is the line `vector<Clonable> v2(bv)`, where a copy of the first vector is created. As shown, the copy keeps intact the actual types of the `Base` references.

At the end of the program, we have created two `Derived1` objects, which are then correctly deleted by the vector's destructors. Here is the full program, illustrating the 'virtual constructor' concept:

```
#include <iostream>
#include <vector>
#include <typeinfo>

class Base
{
    public:
        virtual ~Base();
        virtual Base *clone() const = 0;
};

    inline Base::~Base()
    {}

class Clonable
{
    Base *d_bp;

    public:
        Clonable();
        ~Clonable();
        Clonable(Clonable const &other);
        Clonable &operator=(Clonable const &other);

        // New for virtual constructions:
        Clonable(Base const &bp);
        Base &get() const;

    private:
        void copy(Clonable const &other);
};

    inline Clonable::Clonable()
    :
        d_bp(0)
    {}
    inline Clonable::~Clonable()
    {
        delete d_bp;
    }
    inline Clonable::Clonable(Clonable const &other)
    {
        copy(other);
    }

    Clonable &Clonable::operator=(Clonable const &other)
    {
        if (this != &other)
        {
            delete d_bp;
            copy(other);
```

```
        }
        return *this;
    }

    // New for virtual constructions:
    inline Clonable::Clonable(Base const &bp)
    {
        d_bp = bp.clone();      // allows initialization from
    }                           // Base and derived objects
    inline Base &Clonable::get() const
    {
        return *d_bp;
    }

    void Clonable::copy(Clonable const &other)
    {
        if ((d_bp = other.d_bp))
            d_bp = d_bp->clone();
    }

class Derived1: public Base
{
    public:
        ~Derived1();
        virtual Base *clone() const;
};

    inline Derived::~Derived1()
    {
        std::cout << "~Derived1() called\n";
    }
    inline Base *Derived::clone() const
    {
        return new Derived1(*this);
    }

using namespace std;

int main()
{
    vector<Clonable> bv;

    bv.push_back(Derived1());
    cout << "==\n";

    cout << typeid(bv[0].get()).name() << endl;
    cout << "==\n";

    vector<Clonable> v2(bv);
    cout << typeid(v2[0].get()).name() << endl;
    cout << "==\n";
}
```

Chapter 15

Classes having pointers to members

Classes having `pointer data members` have been discussed in detail in chapter 7. 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 *pointers to members* . Even if the return types and parameter types of member functions are taken into account, surprises are likely to be encountered. For example, consider the following class:

```
class String
{
    char const *(*d_sp)() const;

    public:
        char const *get() const;
};
```

For this class, it is not possible to let a `char const *(*d_sp)() const` data member point to the `get()` member function of the `String` class: `d_sp` cannot be given the address of the member function `get()`.

One of the reasons why this doesn't work is that the variable `d_sp` has global scope, while the member function `get()` is defined within the `String` class, and has `class scope`. The fact that the variable `d_sp` is part of the `String` class is irrelevant. According to `d_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's scope. Doing so, a pointer to a member of the class `String` can be defined as

```
char const *(String::*d_sp)() const;
```

So, due to the `String::` prefix, `d_sp` is defined as a pointer only in the context of the class `String`. It is defined as a pointer to a function in the class `String`, not expecting arguments, not modifying its object's data, and returning a pointer to constant characters.

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::*d_sp)() const` to indicate:

- `d_sp` is a pointer (`*d_sp`),
- to something in the class `String` (`String::*d_sp`).
- It is a pointer to a `const` function, returning a `char const *`: `char const * (String::*d_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 const *`.

Actually, the normal procedure for constructing pointers can still be applied:

- put parentheses around the function name (and its 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::*d_sp)() const
  ```

Another example, this time defining a pointer to a data member. Assume the class `String` contains a `string d_text` member. How to construct a pointer to this member? Again we follow the basic procedure:

- put parentheses around the variable name (and its class name):

  ```
  string (String::d_text)
  ```

- Put a pointer (a star (`*`)) character immediately before the variable-name itself:

  ```
  string (String::*d_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 following pointer to a global function

```
char const * (*sp)() const;
```

becomes a pointer to a member function after prefixing the class-scope:

```
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 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. This can be considered a kind of *relative address*: relative to the object for which the function is called. 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`). In the example itself, note the use of the &-operator to determine the addresses of the members. These operators, as well as the class-scopes are required. Even when used inside the class member implementations themselves:

```
class PointerDemo
{
    public:
        unsigned d_value;
        unsigned get() const;
};

    inline unsigned PointerDemo::get() const
    {
        return d_value;
```

```
    }

int main()
{                                           // initialization
    unsigned (PointerDemo::*getPtr)() const = &PointerDemo::get;
    unsigned PointerDemo::*valuePtr         = &PointerDemo::d_value;

    getPtr   = &PointerDemo::get;           // assignment
    valuePtr = &PointerDemo::d_value;
}
```

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 be given the `PointerDemo` class scope. Pointers to members can also be used with `virtual` member functions. No further changes are required if, e.g., `get()` is defined as a virtual member function.

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 to. With pointers to objects the field selector operator operating on pointers (`->`) or the field selector operating operating on objects (`.`) can be used to select appropriate members.

To use a pointer to member in combination with an object the pointer to member field selector (`.*`) must be 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 other hand, the notion of dereferencing: a dereference operation is used to reach the function or variable the pointer to member points to.

Using the example from the previous section, let's see how we can use the pointer to member function and the pointer to data member:

```
#include <iostream>

class PointerDemo
{
    public:
        unsigned d_value;
        unsigned get() const;
};

    inline unsigned PointerDemo::get() const
    {
        return d_value;
    }

using namespace std;

int main()
```

```
{                                           // initialization
    unsigned (PointerDemo::*getPtr)() const = &PointerDemo::get;
    unsigned PointerDemo::*valuePtr   = &PointerDemo::d_value;

    PointerDemo object;                     // (1) (see text)
    PointerDemo *ptr = &object;

    object.*valuePtr = 12345;               // (2)
    cout << object.*valuePtr << endl;
    cout << object.d_value << endl;

    ptr->*valuePtr = 54321;                 // (3)
    cout << object.d_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. Hence we use the `->*` operator.
- At statement (4) the `.*` and `->*` are used once again, 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 holding a person's name, address and phone number. Let's assume we want to construct a `Person` data base of employees. The employee data base can be queried, but 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 `address()` 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 see all information about the employees, a `SALESPERSON` is allowed to see the employee's phone numbers, while the `CLERK` is only allowed to verify whether a person is actually a member of the organization.

We now construct a member `string personInfo(char const *name)` in the data base class. A standard implementation of this class could be:

```
string PersonData::personInfo(char const *name)
{
    Person *p = lookup(name);   // see if 'name' exists
```

```
        if (!p)
            return "not found";

        switch (d_category)
        {
            case BOARD:
            case SUPERVISOR:
                return allInfo(p);
            case SALESPERSON:
                return noPhone(p);
            case CLERK:
                return nameOnly(p);
        }
    }
```

Although it doesn't take much time, the `switch` must nonetheless be evaluated every time `personCode()` is called. Instead of using a switch, we could define a member `d_infoPtr` as a pointer to a member function of the class `PersonData` returning a `string` and expecting a `Person` reference 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, assuming that the employee status is given as an argument to the constructor of the `PersonData` object.

After having set the `d_infoPtr` member to the appropriate member function, the `personInfo()` member function may now be rewritten:

```
    string PersonData::personInfo(char const *name)
    {
        Person *p = lookup(name);       // see if 'name' exists

        return p ? (this->*d_infoPtr)(p) :  "not found";
    }
```

Note the syntactical construction when using a pointer to member from within a class: `this->*d_infoPtr`.

The member `d_infoPtr` is defined as follows (within the class `PersonData`, omitting other members):

```
    class PersonData
    {
        string (PersonData::*d_infoPtr)(Person *p);
    };
```

Finally, the constructor must initialize `d_infoPtr` to point to the correct 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:
                d_infoPtr = &PersonData::allInfo;
```

```
            case SALESPERSON:
                d_infoPtr = &PersonData::noPhone;
            case CLERK:
                d_infoPtr = &PersonData::nameOnly;
        }
    }
```

Note how addresses of member functions are determined: the class `PersonData` scope *must* be specified, even though we're already inside a member function of the class `PersonData`.

An example using pointers to data members is 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. They exist *separately from* any object of their class. When these static members are public, they can be accessed as global entities, albeit that their class names are required when they are used.

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 usually be accessed like global entities (but see section 10.2.1). 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 associated objects, 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)() = String::n_strings;
                    // address of the  static member function

        cout << (*pfi)() << endl;
                    // print the value produced by String::n_strings()
    }
```

15.5 Pointer sizes

A peculiar characteristic of pointers to members is that their sizes differ from those of 'normal' pointers. Consider the following little program:

```
#include <string>
#include <iostream>

class X
{
    public:
        void fun();
        string d_str;
};
inline void X::fun()
{
    std::cout << "hello\n";
}

using namespace std;

int main()
{
    cout
    << "size of pointer to data-member:     " << sizeof(&X::d_str) << "\n"
    << "size of pointer to member function: " << sizeof(&X::fun) << "\n"
    << "size of pointer to non-member data: " << sizeof(char *) << "\n"
    << "size of pointer to free function:   " << sizeof(&printf) << endl;
}

/*
    generated output:

    size of pointer to data-member:     4
    size of pointer to member function: 8
    size of pointer to non-member data: 4
    size of pointer to free function:   4
*/
```

Note that the size of a pointer to a member function is eight bytes, whereas all other pointers are four bytes (Using the Gnu g++ compiler).

In general, these pointer sizes are not explicitly used, but their differing sizes may cause some confusion in statements like:

```
printf("%p", &X::fun);
```

Of course, `printf()` is likely not the right tool to produce the value of these **C++** specific pointers. The values of these pointers can be inserted into streams when a `union`, reinterpreting the 8-byte pointers as a series of size_t `char` values, is used:

```
#include <string>
#include <iostream>
```

```
#include <iomanip>

class X
{
    public:
        void fun();
        std::string d_str;
};

    inline void X::fun()
    {
        std::cout << "hello\n";
    }

using namespace std;

int main()
{
    union
    {
        void (X::*f)();
        unsigned char *cp;
    }
        u = { &X::fun };

    cout.fill('0');
    cout << hex;
    for (unsigned idx = sizeof(void (X::*)()); idx-- > 0; )
        cout << setw(2) << static_cast<unsigned>(u.cp[idx]);
    cout << endl;
}
```

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 its surrounding class. For example, with the class `string` a type `string::iterator` is available which will provide all characters that are stored in the `string`. This `string::iterator` type could be defined as an object `iterator`, defined as nested class in the class `string`.

A class can be nested in every part of the surrounding class: in the `public, protected` or `private` section. Such a nested class can be considered a member of the surrounding class. The normal access and rules in classes apply to nested classes. If a class is nested in the `public` section of a class, it is visible outside the surrounding class. If it is nested in the `protected` section it is visible in subclasses, derived from the surrounding class (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 special 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
        {
            int d_variable;

            public:
                FirstWithin();
                int var() const;
        };
    private:
        class SecondWithin
        {
            int d_variable;

            public:
                SecondWithin();
                int var() const;
        };
};
```

```
inline int Surround::FirstWithin::var() const
{
    return d_variable;
}
inline int Surround::SecondWithin::var() const
{
    return d_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` therefore has global scope.
- The constructor `FirstWithin()` and the member function `var()` of the class `FirstWithin` are also globally visible.
- The `int d_variable` datamember is only visible to the members of the class `FirstWithin`. Neither the members of `Surround` nor the members of `SecondWithin` can access `d_variable` of the class `FirstWithin` directly.
- The class `SecondWithin` is only visible inside `Surround`. The public members of the class `SecondWithin` can also be used by the members of the class `FirstWithin`, as nested classes can be considered members of their surrounding class.
- The constructor `SecondWithin()` and the member function `var()` of the class `SecondWithin` can also only be reached by the members of `Surround` (and by the members of its nested classes).
- The `int d_variable` datamember of the class `SecondWithin` is only visible to the members of the class `SecondWithin`. Neither the members of `Surround` nor the members of `FirstWithin` can access `d_variable` of the class `SecondWithin` directly.
- As always, an object of the class type is required before its members can be called. This also holds true for nested classes.

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::var()` directly. This is understandable considering the fact that a `Surround` object is not also a `FirstWithin` or `SecondWithin` object. In fact, nested classes are just typenames. It is not implied that objects of such classes automatically exist in the surrounding class. If a member of the surrounding class should use a (non-static) member of a nested class then the surrounding class must define a nested class object, which can thereupon be used by the members of the surrounding class to use members of the nested class.

For example, in the following class definition there is a surrounding class `Outer` and a nested class `Inner`. The class `Outer` contains a member function `caller()` which uses the `inner` object that is composed in `Outer` to call the `infunction()` member function of `Inner`:

```
class Outer
{
    public:
```

```
        void caller();

    private:
        class Inner
        {
            public:
                void infunction();
        };
        Inner d_inner;      // class Inner must be known
};
void Outer::caller()
{
    d_inner.infunction();
}
```

The mentioned 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. On the other hand, the compiler must have seen the definition of the class `Inner` before a data member of that class can be defined.

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 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 they can use any nested class. Even if it appears later in the class interface.

As shown, when (nested) member functions are defined inline, their definition should be put below their class interface. Static nested data members are also normally defined outside of their classes. If the class `FirstWithin` would have a `static size_t` datamember `epoch`, it could be initialized as follows:

```
size_t Surround::FirstWithin::epoch = 1970;
```

Furthermore, multiple scope resolution operators are needed to refer to public static members in code outside of the surrounding class:

```
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, references, parameters or return values to objects of the other nested classes.

For example, the following class `Outer` contains two nested classes `Inner1` and `Inner2`. The class `Inner1` contains a pointer to `Inner2` objects, and `Inner2` contains a pointer to `Inner1` objects. Such cross references require forward declarations. These forward declarations must be specified in the same access-category as their actual definitions. In the following example the `Inner2` forward declaration must be given in a `private` section, as its definition is also part of the class `Outer`'s private interface:

```
class Outer
{
    private:
        class Inner2;       // forward declaration

        class Inner1
        {
            Inner2 *pi2;    // points to Inner2 objects
        };
        class Inner2
        {
            Inner1 *pi1;    // points to Inner1 objects
        };
};
```

16.3 Accessing private members in nested classes

To allow nested classes to access the private members of their surrounding class; to access the private members of other nested classes; or to allow the surrounding class to access the private members of its 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 s_variable`:

```
class Surround
{
    static int s_variable;
    public:
        class FirstWithin
        {
            static int s_variable;
            public:
                int value();
        };
        int value();
    private:
```

```
        class SecondWithin
        {
            static int s_variable;
            public:
                int value();
        };
};
```

If the class `Surround` should be able to access `FirstWithin` and `SecondWithin`'s private members, these latter two classes must declare `Surround` to be their friend. The function `Surround::value()` can thereupon access the private members of its nested classes. For example (note the `friend` declarations in the two nested classes):

```
class Surround
{
    static int s_variable;
    public:
        class FirstWithin
        {
            friend class Surround;
            static int s_variable;
            public:
                int value();
        };
        int value();
    private:
        class SecondWithin
        {
            friend class Surround;
            static int s_variable;
            public:
                int value();
        };
};
inline int Surround::FirstWithin::value()
{
    FirstWithin::s_variable = SecondWithin::s_variable;
    return (s_variable);
}
```

Now, to allow the nested classes access to the private members of their surrounding class, the class `Surround` must declare its 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. When defining members within the class interface implementations of nested class members may use members of the surrounding class that have not yet been seen by the compiler. Finally note that q'`s_variable`' which is defined in the class `Surround` is accessed in the nested classes as `Surround::s_variable`:

```
class Surround
{
    static int s_variable;
    public:
        class FirstWithin
        {
            friend class Surround;
            static int s_variable;
            public:
                int value();
        };
        friend class FirstWithin;
        int value();

    private:
        class SecondWithin
        {
            friend class Surround;
            static int s_variable;
            public:
                int value();
        };
        static void classMember();

        friend class SecondWithin;
};

inline int Surround::value()
{
    FirstWithin::s_variable = SecondWithin::s_variable;
    return s_variable;
}

inline int Surround::FirstWithin::value()
{
    Surround::s_variable = 4;
    Surround::classMember();
    return s_variable;
}

inline int Surround::SecondWithin::value()
{
```

```
        Surround::s_variable = 40;
        return s_variable;
    }
```

Finally, we want to allow the nested classes access to 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 plainly be given in the class `FirstWithin`, as the definition of `SecondWithin` is as yet unknown. A forward declaration of `SecondWithin` is required, and this forward declaration must be provided by the class `Surround`, rather than by 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 `SecondWithin`. Likewise, it is impossible to provide the forward declaration of the nested class `SecondWithin` inside `FirstWithin` as `class Surround::SecondWithin`, 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;
        static int s_variable;
        public:
            class FirstWithin
            {
                friend class Surround;
                friend class SecondWithin;
                static int s_variable;
                public:
                    int value();
            };
            friend class FirstWithin;
            int value();
        private:
            class SecondWithin
            {
                friend class Surround;
                friend class FirstWithin;
                static int s_variable;
                public:
                    int value();
            };
            friend class SecondWithin;
    };
    inline int Surround::value()
    {
        FirstWithin::s_variable = SecondWithin::s_variable;
        return s_variable;
```

```
}

inline int Surround::FirstWithin::value()
{
    Surround::s_variable = SecondWithin::s_variable;
    return s_variable;
}

inline int Surround::SecondWithin::value()
{
    Surround::s_variable = FirstWithin::s_variable;
    return s_variable;
}
```

16.4 Nesting enumerations

Enumerations too may 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`. In the current Gnu **C++** implementation these values are 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
            d_mode;
};
```

Within the class `DataStructure` the values of the `Traversal` enumeration can be used directly. For example:

```
void DataStructure::setTraversal(Traversal mode)
{
    d_mode = mode;
    switch (d_mode)
    {
        forward:
        break;

        backward:
        break;
    }
}
```

Ouside of the class `DataStructure` the name of the enumeration type is not used to refer to the values of the enumeration. Here the classname is sufficient. Only if a variable of the enumeration type is required the name of the enumeration type is needed, as illustrated by the following piece of code:

```
void fun()
{
    DataStructure::Traversal                // enum typename required
        localMode = DataStructure::forward; // enum typename not required

    DataStructure ds;
                                            // enum typename not required
    ds.setTraversal(DataStructure::backward);
}
```

Again, only if `DataStructure` defines a nested class `Nested`, in turn defining the enumeration `Traversal`, the two class scopes are required. In that case the latter example should have been coded as follows:

```
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, irrespective of any value it might represent.

Actually, it is not even 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
{};

using namespace std;

int main()
try
{
    throw EmptyEnum();
}
catch (EmptyEnum)
{
    cout << "Caught empty enum\n";
}
/*
    Generated output:

    Caught empty enum
*/
```

16.5 Revisiting virtual constructors

In section 14.10 the notion of virtual constructors was introduced. In that section a class `Base` was used as an abstract base class. A class `Clonable` was thereupon defined to manage `Base` class pointers in containers like vectors.

As the class `Base` is a very small class, hardly requiring any implementation, it can well be defined as a nested class in `Clonable`. This will emphasize the close relationship that exists between `Clonable` and `Base`, as shown by the way classes are derived from `Base`. One no longer writes:

```
class Derived: public Base
```

but rather:

```
class Derived: public Clonable::Base
```

Other than defining `Base` as a nested class, and deriving from `Clonable::Base` rather than from `Base`, nothing needs to be modified. Here is the program shown earlier in section 14.10, but now using nested classes:

```
#include <iostream>
#include <vector>
#include <typeinfo>

class Clonable
{
    public:
        class Base
        {
            public:
                virtual ~Base();
                virtual Base *clone() const = 0;
        };

    private:
        Base *d_bp;

    public:
        Clonable();
        ~Clonable();
        Clonable(Clonable const &other);
        Clonable &operator=(Clonable const &other);

        // New for virtual constructions:
        Clonable(Base const &bp);
        Base &get() const;

    private:
        void copy(Clonable const &other);
};


inline Clonable::Base::~Base()
{}

inline Clonable::Clonable()
:
    d_bp(0)
{}
inline Clonable::~Clonable()
{
    delete d_bp;
}
inline Clonable::Clonable(Clonable const &other)
{
    copy(other);
}
inline Clonable &Clonable::operator=(Clonable const &other)
{
    if (this != &other)
    {
        delete d_bp;
        copy(other);
    }
```

```
    return *this;
}

inline Clonable::Clonable(Base const &bp)
{
    d_bp = bp.clone();      // allows initialization from
}                           // Base and derived objects

inline Clonable::Base &Clonable::get() const
{
    return *d_bp;
}

inline void Clonable::copy(Clonable const &other)
{
    if ((d_bp = other.d_bp))
        d_bp = d_bp->clone();
}

class Derived1: public Clonable::Base
{
    public:
        ~Derived1();
        virtual Clonable::Base *clone() const;
};

inline Derived1::~Derived1()
{
    std::cout << "~Derived1() called\n";
}
inline Clonable::Base *Derived1::clone() const
{
    return new Derived1(*this);
}

using namespace std;

int main()
{
    vector<Clonable> bv;

    bv.push_back(Derived1());
    cout << "==\n";

    cout << typeid(bv[0].get()).name() << endl;
    cout << "==\n";

    vector<Clonable> v2(bv);
    cout << typeid(v2[0].get()).name() << endl;
    cout << "==\n";
}
```

Chapter 17

The Standard Template Library, generic algorithms

The `Standard Template Library` (STL) is a general purpose library consisting of containers, generic algorithms, iterators, function objects, allocators, adaptors and data structures. The data structures used 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 templates is not discussed (see chapter 18 for that). Rather, this chapter focuses on the *use* of these template algorithms.

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 9.10 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 is 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.

All elements of the STL are defined in the standard namespace. Therefore, a `using namespace std` or comparable directive is required unless it is preferred to specify the required namespace explicitly. This occurs in at least one situation: in header files no `using` directive should be used, so here the `std::` scope specification should always be specified when referring to elements of the STL.

17.1 Predefined function objects

Function objects play important roles in combination with generic algorithms. For example, there exists a generic algorithm `sort()` expecting two iterators defining the range of objects that should be sorted, as well as a function object calling the appropriate comparison operator for two objects. Let's take a quick look at this situation. Assume strings are stored in a vector, and we want to sort the vector in descending order. In that case, sorting the vector stringVec is as simple as:

```
sort(stringVec.begin(), stringVec.end(), greater<std::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. It will call the operator>() of the provided data type (here std::string) whenever its operator()() is called. Eventually, when sort() returns, the first element of the vector will be the greatest element.

The operator()() (function call operator) itself is *not* visible at this point: don't confuse the parentheses in greater<string>() with calling operator()(). When that operator is actually used inside sort(), it receives two arguments: two strings to compare for 'greaterness'. Internally, the operator>() of the data type to which the iterators point (i.e., string) is called by greater<string>'s function operator (operator()()) to compare the two objects. Since greater<>'s function call operator 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 <algorithm>

using namespace std;

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 sort()'s final argument. 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 when called by 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 many commonly used 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. Before predefined function objects can be used the following preprocessor directive must have been specified:

```
#include <functional>
```

Predefined function objects are used predominantly with generic algorithms. Predefined function objects exists for arithmetic, relational, and logical operations. In section 20.4 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. These predefined arithmetic function objects invoke the corresponding operator of the associated data type. For example, for addition the function object `plus<Type>` is available. If we set `type` to `size_t` then the + operator for `size_t` values is used, if we set `type` to `string`, then the + operator for strings is used. For example:

```
#include <iostream>
#include <string>
#include <functional>
using namespace std;

int main(int argc, char **argv)
{
    plus<size_t> uAdd;       // function object to add size_ts

    cout << "3 + 5 = " << uAdd(3, 5) << 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 with all kinds of data types (not only with 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, in turn on each element of an array. 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>
using namespace std;

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>()`. Note here 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 <sstream>
#include <string>
#include <vector>
#include <functional>
#include <numeric>

using namespace std;

class Time
{
    friend ostream &operator<<(ostream &str, Time const &time)
    {
        return cout << time.d_days << " days, " << time.d_hours <<
                                                    " hours, " <<
                        time.d_minutes << " minutes and " <<
                        time.d_seconds << " seconds.";
    }
```

```
    size_t d_days;
    size_t d_hours;
    size_t d_minutes;
    size_t d_seconds;

    public:
        Time(size_t hours, size_t minutes, size_t seconds)
        :
            d_days(0),
            d_hours(hours),
            d_minutes(minutes),
            d_seconds(seconds)
        {}
        Time &operator+=(Time const &rValue)
        {
            d_seconds   += rValue.d_seconds;
            d_minutes   += rValue.d_minutes   + d_seconds / 60;
            d_hours     += rValue.d_hours     + d_minutes / 60;
            d_days      += rValue.d_days      + d_hours   / 24;
            d_seconds   %= 60;
            d_minutes   %= 60;
            d_hours     %= 24;

            return *this;
        }
};
Time const operator+(Time const &lValue, Time const &rValue)
{
    return Time(lValue) += rValue;
}

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 hours, 51 minutes and 30 seconds.
*/
```

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 `Time`'s `operator+=()` 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 a constructor, it defines an insertion operator and it defines its own `operator+()`, adding two time objects.

In `main()` four `Time` objects are stored in a `vector<Time>` object. Then, 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's `operator()()` member calls `operator+()` as a binary operator, passing it its two parameters, returning `operator+()`'s return value.
- `minus<>()`: this object's `operator()()` member calls `operator-()` as a binary operator, passing it its two parameters and returning `operator-()`'s return value.
- `multiplies<>()`: this object's `operator()()` member calls `operator*()` as a binary operator, passing it its two parameters and returning `operator*()`'s return value.
- `divides<>()`: this object's `operator()()` member calls `operator/()`, passing it its two parameters and returning `operator/()`'s return value.
- `modulus<>()`: this object's `operator()()` member calls `operator%()`, passing it its two parameters and returning `operator%()`'s return value.
- `negate<>()`: this object's `operator()()` member calls `operator-()` as a unary operator, passing it its parameter and returning the unary `operator-()`'s return value.

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>
using namespace std;

int main(int argc, char **argv)
{
    int iArr[] = { 1, -2, 3, -4, 5, -6 };

    transform(iArr, iArr + 6, iArr, negate<int>());

    for (int idx = 0; idx < 6; ++idx)
        cout << iArr[idx] << ", ";
```

```
    cout << endl;
}
/*
    Generated output:

    -1, 2, -3, 4, -5, 6,
*/
```

17.1.2 Relational function objects

The relational operators are called by the relational function objects. All standard relational operators are supported: `==, !=, >, >=, <` and `<=`. The following objects are available:

- `equal_to<>()`: this object's `operator()()` member calls `operator==()` as a binary operator, passing it its two parameters and returning `operator==()`'s return value.
- `not_equal_to<>()`: this object's `operator()()` member calls `operator!=()` as a binary operator, passing it its two parameters and returning `operator!=()`'s return value.
- `greater<>()`: this object's `operator()()` member calls `operator>()` as a binary operator, passing it its two parameters and returning `operator>()`'s return value.
- `greater_equal<>()`: this object's `operator()()` member calls `operator>=()` as a binary operator, passing it its two parameters and returning `operator>=()`'s return value.
- `less<>()`: this object's `operator()()` member calls `operator<()` as a binary operator, passing it its two parameters and returning `operator<()`'s return value.
- `less_equal<>()`: this object's `operator()()` member calls `operator<=()` as a binary operator, passing it its two parameters and returning `operator<=()`'s return value.

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>
using namespace std;

int main(int argc, char **argv)
{
    sort(argv, argv + argc, greater_equal<string>());

    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << 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 of the data type to which the iterators point. 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 promotion from `char const *` to `string` is performed silently.

17.1.3 Logical function objects

The logical operators are called by the logical function objects. The standard logical operators are supported: `and`, `or`, and `not`. The following objects are available:

- `logical_and<>()`: this object's `operator()()` member calls `operator&&()` as a binary operator, passing it its two parameters and returning `operator&&()`'s return value.
- `logical_or<>()`: this object's `operator()()` member calls `operator||()` as a binary operator, passing it its two parameters and returning `operator||()`'s return value.
- `logical_not<>()`: this object's `operator()()` member calls `operator!()` as a unary operator, passing it its parameter and returning the unary `operator!()`'s return value.

An example using `operator!()` is provided in the following trivial program, 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>
using namespace std;

int main(int argc, char **argv)
{
    bool bArr[] = {true, true, true, false, false, false};
    size_t const bArrSize = sizeof(bArr) / sizeof(bool);

    for (size_t idx = 0; idx < bArrSize; ++idx)
        cout << bArr[idx] << " ";
    cout << endl;

    transform(bArr, bArr + bArrSize, bArr, logical_not<bool>());

    for (size_t 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)
  ```

  What would such a binder do? First of all, it's a function object, so it needs `operator()()`. Next, it expects two arguments: a reference to another function object and a fixed operand. Although binders are defined as templates, it is illustrative to have a look at their implementations, assuming they were straight functions. Here is such a pseudo-implementation of a binder:

  ```
  class bind2nd
  {
      FunctionObject const &d_object;
      Operand const &d_rvalue;
      public:
          bind2nd(FunctionObject const &object, Operand const &operand);
          ReturnType operator()(Operand const &lvalue);
  };
  inline bind2nd::bind2nd(FunctionObject const &object,
                          Operand const &operand)
  :
      d_object(object),
      d_operand(operand)
  {}
  inline ReturnType bind2nd::operator()(Operand const &lvalue)
  {
      return d_object(lvalue, d_rvalue);
  }
  ```

  When its `operator()()` member is called the binder merely passes the call to the object's `operator()()`, providing it with two arguments: the `lvalue` it itself received and the fixed operand it received via its constructor. Note the simplicity of these kind of classes: all its members can usually be implemented inline.

  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))
```

- *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 used with unary function objects, `not2()` is the negator 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` combined with the `greater()` predicate:

```
count_if(pVector.begin(), pVector.end(),
    bind2nd(not2(greater<Person>()), referencePerson))
```

  Note that `not2()` is a negator negating the truth value of a binary `operator()()` member: it must be used to wrap the binary predicate `greater<Person>()`, negating its truth value.

- Use `not1()` combined with the `bind2nd()` predicate:

```
count_if(pVector.begin(), pVector.end(),
    not1(bind2nd(greater<Person>(), referencePerson)))
```

  Note that `not1()` is a negator negating the truth value of a unary `operator()()` member: it is used to wrap the unary predicate `bind2nd()`, negating its truth value.

  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>
using namespace std;

int main(int argc, char **argv)
{
    int iArr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    cout << count_if(iArr, iArr + 10, bind2nd(less_equal<int>(), 6)) <<
        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
*/
```

One may wonder which of these alternative approaches is fastest. Using the first approach, in which a directly available function object was used, two actions must be performed for each iteration by `count_if()`:

- The binder's `operator()()` is called;
- The operation <= is performed for `int` values.

Using the second approach, in which the `not2` negator is used to negate the truth value of the complementary logical function adaptor, three actions must be performed for each iteration by `count_if()`:

- The binder's `operator()()` is called;
- The negator's `operator()()` is called;
- The operation > is performed for `int` values.

Using the third approach, in which a `not1` negator is used to negate the truth value of the binder, three actions must be performed for each iteration by `count_if()`:

- The negator's `operator()()` is called;
- The binder's `operator()()` is called;
- The operation > is performed for `int` values.

From this, one might deduce that the first approach is fastest. Indeed, using Gnu's `g++` compiler on an old, 166 MHz pentium, performing 3,000,000 `count_if()` calls for each variant, shows the first approach requiring about 70% of the time needed by the other two approaches to complete.

However, these differences disappear if the compiler is instructed to optimize for speed (using the `-O6` compiler flag). When interpreting these results one should keep in mind that multiple nested function calls are merged into a single function call if the implementations of these functions are given inline and if the compiler follows the suggestion to implement these functions as true inline functions indeed. If this is happening, the three approaches all merge to a single operation: the comparison between two `int` values. It is likely that the compiler does so when asked to optimize for speed.

17.2 Iterators

Iterators are objects acting like pointers. Iterators have the following general characteristics:

- Two iterators may be compared for (in)equality using the `==` and `!=` operators. Note that the *ordering* operators (e.g., >, <) normally cannot be used.

- Given an iterator `iter`, `*iter` represents the object the iterator points to (alternatively, `iter->` can be used to reach the members of the object the iterator points to).
- `++iter` or `iter++` advances the iterator to the next element. The notion of advancing an iterator to the next element is consequently applied: several containers have a *reversed_iterator* type, in which the `iter++` operation actually reaches a previous element in a sequence.
- *Pointer arithmetic* may be used with containers having their elements stored consecutively in memory. This includes the `vector` and `deque`. For these containers `iter + 2` points to the second element beyond the one to which `iter` points.
- An interator which is merely defined is comparable to a 0-pointer, as shown by the following little example:

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

int main()
{
    vector<int>::iterator vi;

    cout << &*vi << endl;       // prints 0
}
```

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`. Note that with empty containers the begin- and end-iterators are equal to each other.

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>
using namespace std;

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 series of elements in a constant container. Whereas the elements of the vector in the previous example could have been altered, the elements of the vector in the next example are immutable, and `const_iterators` are required:

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main(int argc, char **argv)
{
    vector<string> const args(argv, argv + argc);

    for
    (
        vector<string>::const_iterator iter = args.begin();
            iter != args.end();
                ++iter
    )
        cout << *iter << " ";
    cout << endl;

    for
    (
        vector<string>::const_reverse_iterator iter = args.rbegin();
            iter != args.rend();
                ++iter
    )
        cout << *iter << " ";
    cout << endl;

    return 0;
}
```

The examples also illustrates that plain pointers can be used instead of 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 characteristics.

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 when describing generic algorithms later in this chapter:

- **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. Notations 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. Analogous notations like these may be observed with other iterator types.

- **OutputIterators**:

  OutputIterators can be used to write to a container. The dereference operator is guaranteed to work as an `lvalue` in expressions, but not necessarily as `rvalue`. Instead of an OutputIterator it is also possible to use, see below, 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 illustrates how to approach 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 defining the range of visited elements, and the third parameter defines the first position where the results of the copy operation should be stored. With the `copy()` algorithm the number of elements that are copied are usually available beforehand, since the number is usually determined using pointer arithmetic. However, there are situations where pointer arithmetic cannot be used. Analogously, the number of resulting elements sometimes differs from the number of elements in the initial range. The generic algorithm `unique_copy()` is a case in point: 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 types of `inserter()` adaptors:

- `back_inserter()`: calls the container's `push_back()` member to add new elements at the end of the container. E.g., to copy all elements of `source` in reversed order to the back of `destination`:

```
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`, shifting existing elements beyond the newly inserted elements:

```
copy(source.begin(), source.end(), inserter(destination,
    destination.begin()));
```

Concentrating on the `back_inserter()`, this iterator expects the name of a container having a member `push_back()`. This member is called by the inserter's `operator()()` member. When a class (other than the abstract containers) supports a `push_back()` container, its objects can also be used as arguments of the `back_inserter()` if the class defines a

```
typedef DataType const &const_reference;
```

in its interface, where `DataType const &` is the type of the parameter of the class's member function `push_back()`. For example, the following program defines a (compilable) skeleton of a class `IntStore`, whose objects can be used as arguments of the `back_inserter` iterator:

```
#include <algorithm>
#include <iterator>
using namespace std;

class Y
{
    public:
        typedef int const &const_reference;
```

```
        void push_back(int const &)
        {}
};

int main()
{
    int arr[] = {1};
    Y y;

    copy(arr, arr + 1, back_inserter(y));
}
```

17.2.2 Iterators for 'istream' objects

The `istream_iterator<Type>()` can be used to define an iterator (pair) for `istream` objects. 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 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 was specified for the begin-iterator is *not* mentioned here.

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>
using namespace std;

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;

    return 0;
}
```

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. The following (non-anonymous) construction could have been used instead of `istream_iterator<string>()`:

```
istream_iterator<string> eos;

copy(istream_iterator<string>(cin), eos, back_inserter(vs));
```

Before `istream_iterators` can be used the following preprocessor directive must have been specified:

```
#include <iterator>
```

This is implied when `iostream` is included.

17.2.3 Iterators for 'istreambuf' objects

Input iterators are also available for `streambuf` objects. Before `istreambuf_iterators` can be used the following preprocessor directive must have been specified:

```
#include <iterator>
```

The `istreambuf_iterator` is available for reading from `streambuf` objects supporting input operations. The standard operations that are 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 `istreambuf_iterator` accessing the `streambuf` of the `istream` object, used as the constructor's argument.

- `istreambuf_iterator<Type>(streambuf *)`:

  This constructor constructs an `istreambuf_iterator` accessing the `streambuf` whose address is used as the constructor's argument.

In section 17.2.4.1 an example is given using both `istreambuf_iterators` and `ostreambuf_iterators`.

17.2.4 Iterators for 'ostream' objects

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:
ostream_iterator<Type> identifier(ostream &outStream, char const *delim);
```

`Type` is the type of the data elements that should be written to the `ostream` stream. `Type` may be any type for which `operator<<()` is defined in combinations 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 following example shows how `istream_iterators` and an `ostream_iterator` may be used 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:

Before `ostream_iterators` can be used the following preprocessor directive must have been specified:

```
#include <iterator>
```

17.2.4.1 Iterators for 'ostreambuf' objects

Before an `ostreambuf_iterator` can be used the following preprocessor directive must have been specified:

```
#include <iterator>
```

The `ostreambuf_iterator` is available for writing to `streambuf` objects supporting output operations. The standard operations that are available for `ostream_iterator` objects are also available for `ostreambuf_iterators`. There are two constructors:

- `ostreambuf_iterator<Type>(ostream)`:

  This constructor constructs an `ostreambuf_iterator` accessing the `streambuf` of the `ostream` object, used as the constructor's argument, to insert values of type `Type`.

- `ostreambuf_iterator<Type>(streambuf *)`:

  This constructor constructs an `ostreambuf_iterator` accessing the `streambuf` whose address is used as the constructor's argument.

Here is an example using both `istreambuf_iterators` and an `ostreambuf_iterator`, showing yet another way to copy a stream:

```
#include <iostream>
```

```
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    istreambuf_iterator<char> in(cin.rdbuf());
    istreambuf_iterator<char> eof;
    ostreambuf_iterator<char> out(cout.rdbuf());

    copy(in, eof, out);

    return 0;
}
```

17.3 The class 'auto_ptr'

One of the problems using pointers is that strict bookkeeping is required about their memory use and lifetime. 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()`: the allocated `int` value remains inaccessibly allocated:

```
void fun()
{
    new int;
}
```

To prevent memory leaks strict bookkeeping is required: 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;
}
```

Now `fun()` only wastes a bit of time.

When a pointer variable points to *a single value or object*, the bookkeeping requirements may be relaxed when the pointer variable is defined as a `std::auto_ptr` object. Auto_ptrs are *objects*, masquerading as pointers. Since they're objects, their destructors are called when they go out of scope, and because of that, their destructors will take the responsibility of deleting the dynamically allocated memory.

Before `auto_ptrs` can be used the following preprocessor directive must have been specified:

```
#include <memory>
```

Normally, an `auto_ptr` object is initialized using a dynamically created value or object.

The following *restrictions* apply to auto_ptrs:

- 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. The auto_ptr's interface was designed to prevent this from happening. Once an auto_ptr object goes out of scope, it deletes the memory it points to, immediately changing any other object also pointing to the allocated memory into a wild pointer.

The class auto_ptr defines several member functions 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 next 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 angle 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 point to a block of memory 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 using a copy constructor:

  ```
  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 allocated by operator new operator. The generic form is:

```
auto_ptr<type> identifier(new-expression);
```

For example, to initialize an auto_ptr to point to a string object the following construction can be used:

```
auto_ptr<string> strPtr(new string("Hello world"));
```

To initialize an `auto_ptr` to point to a `double` value the following construction can be used:

```
auto_ptr<double> dPtr(new double(123.456));
```

Note the use of `operator new` in the above expressions. Using `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 as used in the `new` expression.

In the example allocating an `int` values given in section 17.3, the memory leak can be avoided using an `auto_ptr` object:

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

void fun()
{
    auto_ptr<int> ip(new int);
}
```

All member functions available for objects 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 <memory>
using namespace std;

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'

An `auto_ptr` may also be initialized by 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<string>`, given the variable `sp` defined in the previous section, the following construction can be used:

```
auto_ptr<string> strPtr(sp);
```

Analogously, the assignment operator can be used. An `auto_ptr` object may be assigned to another `auto_ptr` object of the same type. For example:

```
#include <iostream>
#include <memory>
#include <string>
using namespace std;

int main()
{
    auto_ptr<string> hello1(new string("Hello world"));
    auto_ptr<string> hello2(hello1);
    auto_ptr<string> 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. This effectively changes `hello1` into a 0-pointer.
- Then `hello3` is defined as a default `auto_ptr<string>`, but it receives its value through an assignment from `hello2`, which then becomes a 0-pointer too.

The program generates a *segmentation fault*. The reason for this will now be clear: it is caused by dereferencing 0-pointers. At the end, only `hello3` actually points to a `string`.

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 not pointing 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 next section.

17.3.5 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, and turns into a 0-pointer.

- `Type &auto_ptr<Type>operator*()`:

  This operator returns a reference to the information stored in the `auto_ptr` object. It acts like a normal pointer dereference operator.

- `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 an 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 (and changes into a 0-pointer). The member can be used to transfer the information stored in the `auto_ptr` object to a plain `Type` pointer. It is the responsibility of the programmer to delete the memory returned by this member function.

- `void auto_ptr<Type>::reset(Type *)`:

  This operator may also be called *without* argument, to delete the memory stored in the `auto_ptr` object, or with a pointer to a dynamically allocated block of memory, which will thereupon be the memory accessed by the `auto_ptr` object. This member function can be used to assign a new block of memory (new content) to an `auto_ptr` object.

17.3.6 Constructors and pointer data members

Now that the `auto_ptr`'s main features have been described, consider the following simple class:

```
// required #includes

class Map
{
    std::map<string, Data> *d_map;
    public:
        Map(char const *filename) throw(std::exception);
};
```

The class's constructor `Map()` performs the following tasks:

- It allocates a `std::map` object;
- It opens the file whose name is given as the constructor's argument;
- It reads the file, thereby filling the map.

Of course, it may not be possible to open the file. In that case an appropriate exception is thrown. So, the constructor's implementation will look somewhat like this:

```
Map::Map(char const *fname)
:
    d_map(new std::map<std::string, Data>) throw(std::exception)
{
    ifstream istr(fname);
    if (!istr)
        throw std::exception("can't open the file");
    fillMap(istr);
}
```

What's wrong with this implementation? Its main weakness is that it hosts a potential memory leak. The memory leak only occurs when the exception is actually thrown. In all other cases, the function operates perfectly well. When the exception is thrown, the map has just been dynamically allocated. However, even though the class's destructor will dutifully call `delete d_map`, the destructor is actually never called, as the destructor will only be called to destroy objects that were constructed completely. Since the constructor terminates in an exception, its associated object is not constructed completely, and therefore that object's destructor is never called.

`Auto_ptrs` may be used to prevent these kinds of problems. By defining `d_map` as

```
    std::auto_ptr<std::map<std::string, Data> >
```

it suddenly changes into an object. Now, `Map`'s constructor may safely throw an exception. As `d_map` is an object itself, its destructor will be called by the time the (however incompletely constructed) `Map` object goes out of scope.

As a rule of thumb: classes should use `auto_ptr` objects, rather than plain pointers for their pointer data members if there's any chance that their constructors will end prematurely in an exception.

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 file;
- The function prototype;
- A short description;
- A short example.

In the prototypes of the algorithms `Type` is used to specify a generic data type. Also, the particular type of iterator (see section 17.2) that is required is mentioned, as well as other generic types that might be required (e.g., performing `BinaryOperations`, like `plus<Type>()`).

Almost every generic algorithm expects 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, function objects used by the algorithms usually receive `Type const &` objects or values: function objects can therefore not modify the objects they receive as their arguments. This does not hold true for *modifying generic algorithms*, which *are* (of course) able to modify the objects they operate upon.

Generic algorithms may be categorized. 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(); reverse_copy(); rotate_copy(); 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>`
  accumulate(); adjacent_difference(); inner_product(); partial_sum();

- Searchers: performing search (and find) operations:

  Requires: `#include <algorithm>`
  adjacent_find(); binary_search(); equal_range(); find(); find_end(); find_first_of(); find_if(); lower_bound(); max_element(); min_element(); search(); search_n(); set_difference(); set_intersection(); set_symmetric_difference(); set_union(); upper_bound();

- Shufflers: performing reordering operations (sorting, merging, permuting, shuffling, swapping):

  Requires: `#include <algorithm>`
  inplace_merge(); iter_swap(); merge(); next_permutation(); nth_element(); partial_sort(); partial_sort_copy(); partition(); prev_permutation(); random_shuffle(); remove(); remove_copy(); remove_copy_if(); remove_if(); reverse(); reverse_copy(); rotate(); rotate_copy(); sort(); stable_partition(); stable_sort(); swap(); unique();

- Visitors: visiting elements in a range:

  Requires: `#include <algorithm>`
  for_each(); replace(); replace_copy(); replace_copy_if(); replace_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>
using namespace std;
```

```
int main()
{
    int         ia[] = {1, 2, 3, 4};
    vector<int> iv(ia, ia + 4);

    cout <<
        "Sum of values: " << accumulate(iv.begin(), iv.end(), int()) <<
        endl <<
        "Product of values: " << accumulate(iv.begin(), iv.end(), int(1),
                                        multiplies<int>()) << endl;

    return 0;
}
/*
    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>
using namespace std;

int main()
{
```

```
    int             ia[] = {1, 2, 5, 10};
    vector<int>     iv(ia, ia + 4);
    vector<int>     ov(iv.size());

    adjacent_difference(iv.begin(), iv.end(), ov.begin());

    copy(ov.begin(), ov.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    adjacent_difference(iv.begin(), iv.end(), ov.begin(), minus<int>());

    copy(ov.begin(), ov.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    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
{
    size_t d_minimum;

    public:
```

```
        SquaresDiff(size_t minimum)
        :
            d_minimum(minimum)
        {}
        bool operator()(size_t first, size_t second)
        {
            return second * second - first * first  >= d_minimum;
        }
};

using namespace std;

int main()
{
    string sarr[] =
        {
            "Alpha", "bravo", "charley", "delta", "echo", "echo",
            "foxtrot", "golf"
        };
    string *last = sarr + sizeof(sarr) / sizeof(string);
    string *result = adjacent_find(sarr, last);

    cout << *result << endl;
    result = adjacent_find(++result, last);

    cout << "Second time, starting from the next position:\n" <<
        (
            result == last ?
                "** No more adjacent equal elements **"
            :
                "*result"
        ) << endl;

    size_t iv[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    size_t *ilast = iv + sizeof(iv) / sizeof(size_t);
    size_t *ires = adjacent_find(iv, ilast, SquaresDiff(10));

    cout <<
        "The first numbers for which the squares differ at least 10: "
        << *ires << " and " << *(ires + 1) << endl;

    return 0;
}
/*
    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>
using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "bravo", "charley", "delta", "echo",
            "foxtrot", "golf", "hotel"
        };
    string *last = sarr + sizeof(sarr) / sizeof(string);
    bool result = binary_search(sarr, last, "foxtrot");

    cout << (result ? "found " : "didn't find ") << "foxtrot" << 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;

    return 0;
}
```

```
/*
    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)` is copied to an output range, starting at `destination`, using the assignment operator of the underlying data type. The return value is the OutputIterator pointing just beyond the last element that was copied to the destination range (so, 'last' in the destination range is returned).
- Example:
  Note the second call to `copy()`. It uses an `ostream_iterator` for `string` objects. This iterator will write the `string` values to the specified `ostream` (i.e., `cout`), separating the values by the specified separation string (i.e., `" "`).

```
#include <algorithm>
#include <string>
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "bravo", "charley", "delta", "echo",
            "foxtrot", "golf", "hotel"
        };
    string  *last = sarr + sizeof(sarr) / sizeof(string);

    copy(sarr + 2, last, sarr); // move all elements two positions left

                                // copy to cout using an ostream_iterator
                                // for strings,
    copy(sarr, last, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
```

```
/*
    Generated output:

    charley delta echo 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 to the last element that was copied to the destination range (so, 'first' in the destination range, pointed to by `last2 - (last - first)`, is returned).
- Example:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "bravo", "charley", "delta", "echo",
            "foxtrot", "golf", "hotel"
        };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        copy_backward(sarr + 3, last, last - 3),
        last,
        ostream_iterator<string>(cout, " ")
    );
    cout << endl;
```

```
    return 0;
}
/*
    Generated output:

    golf hotel foxtrot golf hotel foxtrot golf hotel
*/
```

17.4.7 count()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `size_t count(InputIterator first, InputIterator last, Type const &value);`
- Description:
  - The number of times `value` occurs in the iterator range `[first, last)` is returned. To determine whehter `value` is equal to an element in the iterator range `Type::operator==()` is used.
- Example:

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

int main()
{
    int ia[] = {1, 2, 3, 4, 3, 4, 2, 1, 3};

    cout << "Number of times the value 3 is available: " <<
        count(ia, ia + sizeof(ia) / sizeof(int), 3) <<
        endl;

    return 0;
}
/*
    Generated output:

    Number of times the value 3 is available: 3
*/
```

17.4.8 count_if()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `size_t count_if(InputIterator first, InputIterator last, Predicate predicate);`
- Description:
  - The number of times unary predicate '`predicate`' returns `true` when applied to the elements implied by the iterator range `[first, last)` is returned.
- Example:

```
#include <algorithm>
#include <iostream>

class Odd
{
    public:
        bool operator()(int value)
        {
            return value & 1;
        }
};

using namespace std;

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;

    return 0;
}
/*
    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()(std::string const &first,
                        std::string const &second) const
        {
            return !strcasecmp(first.c_str(), second.c_str());
        }
};

using namespace std;

int main()
{
    string first[] =
        {
            "Alpha", "bravo", "Charley", "delta", "Echo",
            "foxtrot", "Golf", "hotel"
        };
    string second[] =
        {
            "alpha", "bravo", "charley", "delta", "echo",
            "foxtrot", "golf", "hotel"
        };
    string *last = first + sizeof(first) / sizeof(string);

    cout << "The elements of ‘first’ and ‘second’ are pairwise " <<
        (equal(first, last, second) ? "equal" : "not equal") <<
        endl <<
        "compared case-insensitively, they are " <<
        (
            equal(first, last, second, CaseString()) ?
                "equal" : "not equal"
        ) << endl;

    return 0;
}
```

```
/*
    Generated output:

    The elements of 'first' and 'second' are pairwise not equal
    compared case-insensitively, they are equal
*/
```

17.4.10 equal_range()

- Header file:

```
#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 12.3.6) and `multimap` (section 12.3.7)):
  - The first prototype: starting from a sorted sequence (where the `operator<()` of the data type to which the iterators point was used to sort the elements in the provided range), a pair of iterators is returned representing the return value of, respectively, `lower_bound()` (returning the first element that is not smaller than the provided reference value, see section 17.4.25) and `upper_bound()`(returning the first element beyond the provided reference value, see section 17.4.66).
  - The second prototype: starting from a sorted sequence (where the `comp` function object was used to sort the elements in the provided range), a pair of iterators is returned representing the return values of, respectively, the functions `lower_bound()` (section 17.4.25) and `upper_bound()`(section 17.4.66).
- Example:

```
#include <algorithm>
#include <functional>
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
    int                 range[] = {1, 3, 5, 7, 7, 9, 9, 9};
    size_t const      size = sizeof(range) / sizeof(int);

    pair<int *, int *>  pi;

    pi = equal_range(range, range + size, 6);

    cout << "Lower bound for 6: " << *pi.first << endl;
    cout << "Upper bound for 6: " << *pi.second << endl;
```

```
    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;

    return 0;
}
/*
    Generated output:

            Lower bound for 6: 7
            Upper bound for 6: 7
            Lower bound for 7: 7 7 9 9 9
            Upper bound for 7: 9 9 9
            Sorted in descending order
            9 9 9 7 7 5 3 1
            Lower bound for 7: 7 7 5 3 1
            Upper bound for 7: 5 3 1
*/
```

17.4.11 fill()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `void fill(ForwardIterator first, ForwardIterator last, Type const &value);`
- Description:

  - all the elements implied by the iterator range `[first, last)` are initialized to `value`, overwriting the previous stored values.

- Example:

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
    vector<int>     iv(8);

    fill(iv.begin(), iv.end(), 8);

    copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    8 8 8 8 8 8 8 8
*/
```

17.4.12 fill_n()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `void fill_n(ForwardIterator first, Size n, Type const &value);`
- Description:
  - `n` elements starting at the element pointed to by `first` are initialized to `value`, overwriting the previous stored values.
- Example:

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
    vector<int>     iv(8);
```

```
        fill_n(iv.begin() + 2, 4, 8);

        copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
        cout << endl;

        return 0;
    }
    /*
        Generated output:

        0 0 8 8 8 8 0 0
    */
```

17.4.13 find()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `InputIterator find(InputIterator first, InputIterator last, Type const &value);`
- Description:
  - Element `value` is searched for in the range of the elements implied by the iterator range `[first, last)`. An iterator pointing to the first element found is returned. If the element was not found, `last` is returned. The `operator==()` of the underlying data type is used to compare the elements.
- Example:

```
#include <algorithm>
#include <string>
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "bravo", "charley", "delta", "echo"
        };
    string  *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        find(sarr, last, "delta"), last, ostream_iterator<string>(cout, " ")
    );
    cout << 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;
        }

        return 0;

    }
    /*
        Generated output:

        delta echo
        'india' was not found in the range
        alpha bravo charley delta echo
    */
```

17.4.14 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 <iterator>
#include <iostream>

class Twice
{
    public:
        bool operator()(size_t first, size_t second) const
        {
```

```
            return first == (second << 1);
        }
};

using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "bravo", "charley", "delta", "echo",
            "foxtrot", "golf", "hotel",
            "foxtrot", "golf", "hotel",
            "india", "juliet", "kilo"
        };
    string search[] =
        {
            "foxtrot",
            "golf",
            "hotel"
        };
    string  *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        find_end(sarr, last, search, search + 3),   // sequence starting
        last, ostream_iterator<string>(cout, " ")   // at 2nd 'foxtrot'
    );
    cout << endl;

    size_t range[] = {2, 4, 6, 8, 10, 4, 6, 8, 10};
    size_t nrs[]   = {2, 3, 4};

    copy                // sequence of values starting at last sequence
    (                   // of range[] that are twice the values in nrs[]
        find_end(range, range + 9, nrs, nrs + 3, Twice()),
        range + 9, ostream_iterator<size_t>(cout, " ")
    );
    cout << endl;

    return 0;
}
/*
    Generated output:

    foxtrot golf hotel india juliet kilo
    4 6 8 10
*/
```

17.4.15 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 <iterator>
#include <iostream>

class Twice
{
    public:
        bool operator()(size_t first, size_t second) const
        {
            return first == (second << 1);
        }
};

using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "bravo", "charley", "delta", "echo",
            "foxtrot", "golf", "hotel",
            "foxtrot", "golf", "hotel",
            "india", "juliet", "kilo"
        };
    string search[] =
        {
```

```
            "foxtrot",
            "golf",
            "hotel"
        };
    string  *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (                                               // sequence starting
        find_first_of(sarr, last, search, search + 3),  // at 1st 'foxtrot'
        last, ostream_iterator<string>(cout, " ")
    );
    cout << endl;

    size_t range[] = {2, 4, 6, 8, 10, 4, 6, 8, 10};
    size_t 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<size_t>(cout, " ")
    );
    cout << endl;

    return 0;
}
/*
    Generated output:

    foxtrot golf hotel foxtrot golf hotel india juliet kilo
    4 6 8 10 4 6 8 10
*/
```

17.4.16 find_if()

- Header file:

```
    #include <algorithm>
```

- Function prototype:
  - `InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);`
- Description:
  - An iterator pointing to the first element in the range implied by the iterator range `[first, last)` for which the (unary) predicate `pred` returns `true` is returned. If the element was not found, `last` is returned.
- Example:

```
#include <algorithm>
#include <string>
#include <iterator>
#include <iostream>
```

```
class CaseName
{
    std::string  d_string;

    public:
        CaseName(char const *str): d_string(str)
        {}
        bool operator()(std::string const &element)
        {
            return !strcasecmp(element.c_str(), d_string.c_str());
        }
};

using namespace std;

int main()
{
    string sarr[] =
        {
            "Alpha", "Bravo", "Charley", "Delta", "Echo",
        };
    string  *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        find_if(sarr, last, CaseName("charley")),
        last, ostream_iterator<string>(cout, " ")
    );
    cout << 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;
    }

    return 0;

}
/*
    Generated output:

    Charley Delta Echo
    'india' was not found in the range
    Alpha Bravo Charley Delta Echo
*/
```

17.4.17 for_each()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `Function for_each(ForwardIterator first, ForwardIterator last, Function func);`
- Description:
  - Each of the elements implied by the iterator range `[first, last)` is passed in turn as a reference to the function (or function object) `func`. The function may modify the elements it receives (as the used iterator is a forward iterator). Alternatively, if the elements should be transformed, `transform()` (see section 17.4.63) can be used. The function itself or a copy of the provided function object is returned: see the example below, in which an extra argument list is added to the `for_each()` call, which argument is eventually also passed to the function given to `for_each()`. Within `for_each()` the return value of the function that is passed to it is ignored.
- Example:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <cctype>

void lowerCase(char &c)                     // ‘c’ *is* modified
{
    c = static_cast<char>(tolower(c));
}
                                            // ‘str’ is *not* modified
void capitalizedOutput(std::string const &str)
{
    char    *tmp = strcpy(new char[str.size() + 1], str.c_str());

    std::for_each(tmp + 1, tmp + str.size(), lowerCase);

    tmp[0] = toupper(*tmp);
    std::cout << tmp << " ";
    delete tmp;
};

using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "BRAVO", "charley", "DELTA",  "echo",
            "FOXTROT", "golf", "HOTEL",
        };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    for_each(sarr, last, capitalizedOutput)("that’s all, folks");
    cout << endl;

    return 0;
```

```
}
/*
    Generated output:

    Alpha Bravo Charley Delta Echo Foxtrot Golf Hotel That’s all, folks
*/
```

- Here is another example, using a function object:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <cctype>

void lowerCase(char &c)
{
    c = tolower(c);
}

class Show
{
    int d_count;

    public:
        Show()
        :
            d_count(0)
        {}

        void operator()(std::string &str)
        {
            std::for_each(str.begin(), str.end(), lowerCase);
            str[0] = toupper(str[0]);   // here assuming str.length()
            std::cout << ++d_count << " " << str << "; ";
        }

        int count() const
        {
            return d_count;
        }
};

using namespace std;

int main()
{
    string sarr[] =
        {
            "alpha", "BRAVO", "charley", "DELTA", "echo",
            "FOXTROT", "golf", "HOTEL",
        };
    string *last = sarr + sizeof(sarr) / sizeof(string);

    cout << for_each(sarr, last, Show()).count() << endl;
```

```
    return 0;
}
/*
    Generated output (all on a single line):

    1 Alpha; 2 Bravo; 3 Charley; 4 Delta; 5 Echo; 6 Foxtrot;
                                               7 Golf; 8 Hotel; 8
*/
```

The example also shows that the `for_each` algorithm may be used with functions defining `const` and non-`const` parameters. Also, see section 17.4.63 for differences between the `for_each()` and `transform()` generic algorithms.

The `for_each()` algorithm cannot directly be used (i.e., by passing `*this` as the function object argument) inside a member function to modify its own object as the `for_each()` algorithm first creates its own copy of the passed function object. A *wrapper class* whose constructor accepts a pointer or reference to the current object and possibly to one of its member functions solves this problem. In section 20.7 the construction of such wrapper classes is described.

17.4.18 generate()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `void generate(ForwardIterator first, ForwardIterator last, Generator generator);`
- Description:
  - All elements implied by the iterator range `[first, last)` are initialized by the return value of `generator`, which can be a function or function object. `Generator::operator()()` does not receive any arguments. The example uses a well-known fact from algebra: in order to obtain the square of `n + 1`, add `1 + 2 * n` to `n * n`.
- Example:

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream>

class NaturalSquares
{
    size_t d_newsqr;
    size_t d_last;

    public:
        NaturalSquares(): d_newsqr(0), d_last(0)
        {}
        size_t operator()()
        {                   // using: (a + 1)^2 == a^2 + 2*a + 1
```

```
            return d_newsqr += (d_last++ << 1) + 1;
        }
};

using namespace std;

int main()
{
    vector<size_t>    uv(10);

    generate(uv.begin(), uv.end(), NaturalSquares());

    copy(uv.begin(), uv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    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 <iterator>
#include <iostream>

class NaturalSquares
{
    size_t d_newsqr;
    size_t d_last;

    public:
        NaturalSquares(): d_newsqr(0), d_last(0)
        {}
        size_t operator()()
        {                           // using: (a + 1)^2 == a^2 + 2*a + 1
```

```
                return d_newsqr += (d_last++ << 1) + 1;
            }
    };

    using namespace std;

    int main()
    {
        vector<size_t>    uv(10);

        generate_n(uv.begin(), 5, NaturalSquares());

        copy(uv.begin(), uv.end(), ostream_iterator<int>(cout, " "));
        cout << endl;

        return 0;
    }
    /*
        Generated output:

        1 4 9 16 25 0 0 0 0 0
    */
```

17.4.20 includes()

- Header file:

```
#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 data type to which the iterators point. 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 seqence `[first1, second1)` (the second range is a subset of the first range).
- Example:

```
#include <algorithm>
#include <string>
#include <iostream>
```

```
class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return !strcasecmp(first.c_str(), second.c_str());
        }
};

using namespace std;

int main()
{
    string first1[] =
        {
            "alpha", "bravo", "charley", "delta",  "echo",
            "foxtrot", "golf", "hotel"
        };
    string first2[] =
        {
            "Alpha", "bravo", "Charley", "delta",  "Echo",
            "foxtrot", "Golf", "hotel"
        };
    string second[] =
        {
            "charley", "foxtrot", "hotel"
        };
    size_t n = sizeof(first1) / sizeof(string);

    cout << "The elements of ‘second’ are " <<
        (includes(first1, first1 + n, second, second + 3) ? "" : "not")
        << " contained in the first sequence:\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";

    return 0;
}
/*
    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 file:

```
#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 data type to which the iterators point.
  - The second prototype: binary operator `op1` instead of the default addition operator, and binary operator `op2` instead of the default multiplication operator are applied to all pairwise elements implied by the range `[first1, last1)` and the same number of elements starting at the element pointed to by `first2`. The final result is returned.
- Example:

```
#include <numeric>
#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>

class Cat
{
    std::string  d_sep;
    public:
        Cat(std::string const &sep)
        :
            d_sep(sep)
        {}
        std::string operator()
            (std::string const &s1, std::string const &s2) const
        {
            return s1 + d_sep + s2;
```

```
        }
};

using namespace std;

int main()
{
    size_t ia1[] = {1, 2, 3, 4, 5, 6, 7};
    size_t ia2[] = {7, 6, 5, 4, 3, 2, 1};
    size_t init = 0;

    cout << "The sum of all squares in ";
    copy(ia1, ia1 + 7, ostream_iterator<size_t>(cout, " "));
    cout << "is " <<
        inner_product(ia1, ia1 + 7, ia1, init) << endl;

    cout << "The sum of all cross-products in ";
    copy(ia1, ia1 + 7, ostream_iterator<size_t>(cout, " "));
    cout << " and ";
    copy(ia2, ia2 + 7, ostream_iterator<size_t>(cout, " "));
    cout << "is " <<
        inner_product(ia1, ia1 + 7, ia2, init) << endl;

    string names1[] = {"Frank", "Karel", "Piet"};
    string names2[] = {"Brokken", "Kubat", "Plomp"};

    cout << "A list of all combined names in ";
    copy(names1, names1 + 3, ostream_iterator<string>(cout, " "));
    cout << "and\n";
    copy(names2, names2 + 3, ostream_iterator<string>(cout, " "));
    cout << "is:" <<
        inner_product(names1, names1 + 3, names2, string("\t"),
            Cat("\n\t"), Cat(" ")) <<
        endl;

    return 0;
}
/*
    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 file:

```
#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 data type to which the iterators point). The final series is stored in the range `[first, last)`.
  - The second prototype: the two (sorted) ranges `[first, middle)` and `[middle, last)` are merged, keeping a sorted list (using the boolean result of the binary comparison operator `comp`). The final series is stored in the range `[first, last)`.
- Example:

```
#include <algorithm>
#include <string>
#include <iterator>
#include <iostream>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(first.c_str(), second.c_str()) < 0;
        }
};

using namespace std;

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;

        return 0;
    }
    /*
        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 prototype:
  - `void iter_swap(ForwardIterator1 iter1, ForwardIterator2 iter2);`
- Description:
  - The elements pointed to by `iter1` and `iter2` are swapped.
- Example:

```
#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string first[] = {"alpha", "bravo", "charley"};
    string second[] = {"echo", "foxtrot", "golf"};
    size_t const n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    for (size_t idx = 0; idx < n; ++idx)
        iter_swap(first + idx, second + idx);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;
```

```
    return 0;
}
/*
    Generated output:

    Before:
    alpha bravo charley
    echo foxtrot golf
    After:
    echo foxtrot golf
    alpha bravo charley
*/
```

17.4.24 lexicographical_compare()

- Header file:

```
#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 `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. So, `false` is returned:
    - at the first element in the first range which is greater than the corresponding element in the second range (using `operator<()` of the data type to which the iterators point, reversing the operands),
    - 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 `operator<()` of the data type to which the iterators point.
- Example:

```
#include <algorithm>
#include <iterator>
#include <iostream>
#include <string>

class CaseString
{
```

```
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(first.c_str(), second.c_str()) < 0;
        }
};

using namespace std;

int main()
{
    string word1 = "hello";
    string word2 = "help";

    cout << word1 << " is " <<
        (
            lexicographical_compare(word1.begin(), word1.end(),
                                    word2.begin(), word2.end()) ?
                "before "
            :
                "beyond or at "
        ) <<
        word2 << " in the alphabet\n";

    cout << word1 << " is " <<
        (
            lexicographical_compare(word1.begin(), word1.end(),
                                    word1.begin(), word1.end()) ?
                "before "
            :
                "beyond or at "
        ) <<
        word1 << " in the alphabet\n";

    cout << word2 << " is " <<
        (
            lexicographical_compare(word2.begin(), word2.end(),
                                    word1.begin(), word1.end()) ?
                "before "
            :
                "beyond or at "
        ) <<
        word1 << " in the alphabet\n";

    string one[] = {"alpha", "bravo", "charley"};
    string two[] = {"ALPHA", "BRAVO", "DELTA"};

    copy(one, one + 3, ostream_iterator<string>(cout, " "));
    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";

        return 0;
    }
    /*
        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 file:

```
#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 data type to which the iterators point 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 `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 `false` is returned. If no such element is found, `last` is returned.
- Example:

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <functional>
using namespace std;

int main()
```

```
{
    int     ia[] = {10, 20, 30};

    cout << "Sequence: ";
    copy(ia, ia + 3, ostream_iterator<int>(cout, " "));
    cout << 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;

    return 0;
}
/*
    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 data type to which the iterators point.

  - 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()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(second.c_str(), first.c_str()) > 0;
        }
};

using namespace std;

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";

    return 0;
}
/*
    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 data type to which the iterators point 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);
        }
};

using namespace std;

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;

    return 0;
}
/*
    Generated output:

    The max. int value is 10
    The max. absolute int value is -12
*/
```

17.4.28 merge()

- Header file:

```
#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 data type to which the iterators point). The final series is stored in the range starting at `result` and ending just before the `OutputIterator` returned by the function.
  - The first 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` returned by the function.

- Example:

```
#include <algorithm>
#include <string>
#include <iterator>
#include <iostream>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(first.c_str(), second.c_str()) < 0;
        }
};

using namespace std;

int main()
{
    string range1[] =
        {                                       // 5 elements
            "alpha", "bravo", "foxtrot", "hotel", "zulu"
        };
    string range2[] =
        {                                       // 4 elements
            "echo", "delta", "golf", "romeo"
        };
    string 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"
```

```
        };
    string range4[] =
        {
            "delta", "ECHO", "GOLF", "romeo"
        };

    copy(result,
        merge(range3, range3 + 5, range4, range4 + 4, result,
                                                          CaseString()),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    alpha bravo echo delta 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 data type to which the iterators point.
  - 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()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(second.c_str(), first.c_str()) > 0;
        }
};
```

```
using namespace std;

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";

    return 0;
}
/*
    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, using `operator<()` of the data type to which the iterators point.
  - The second prototype: rather than using `operator<()`, the binary predicate `comp` is used to make the comparisons between the elements implied by the iterator range `[first, last)`. The element for which `comp` returns `false` most often is returned.
- Example:

```
#include <algorithm>
#include <iostream>

class AbsValue
{
    public:
        bool operator()(int first, int second) const
```

```
        {
            return abs(first) < abs(second);
        }
};

using namespace std;

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;

    return 0;
}
/*
    Generated output:

    The minimum int value is -12
    The minimum absolute int value is -2
*/
```

17.4.31 mismatch()

- Header file:

```
#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 data type to which the iterators point. Comparison stops if the compared elements differ (i.e., `operator==()` returns false) or `last1` is reached. A `pair` containing iterators pointing to the final positions is returned. The second sequence may contain more elements than the first sequence. The behavior of the algorithm is undefined if the second sequence contains fewer elements than the first sequence.
  - The second prototype: the two sequences of elements starting at `first1` and `first2` are compared using the binary comparison operation as defined by `comp`, instead of `operator==()`. Comparison stops if the `comp` function returns `false` or `last1` is reached. A `pair` containing iterators pointing to the final positions is returned. The second sequence may contain more elements than the first sequence. The behavior of the algorithm is undefined if the second sequence contains fewer elements than the first sequence.

- Example:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <utility>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(first.c_str(), second.c_str()) == 0;
        }
};

using namespace std;

int main()
{
    string range1[] =
        {
            "alpha", "bravo", "foxtrot", "hotel", "zulu"
        };
    string range2[] =
        {
            "alpha", "bravo", "foxtrot", "Hotel", "zulu"
        };
    pair<string *, string *> pss = mismatch(range1, range1 + 5, range2);

    cout << "The elements " << *pss.first << " and " << *pss.second <<
            " at offset " << (pss.first - range1) << " differ\n";
    if
    (
        mismatch(range1, range1 + 5, range2, CaseString()).first
        ==
        range1 + 5
    )
        cout << "When compared case-insensitively they match\n";

    return 0;
}
/*
    Generated output:

    The elements hotel and Hotel at offset 3 differ
    When compared case-insensitively they match
*/
```

17.4.32 next_permutation()

- Header file:

```
#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()` reorders the following series:

    ```
    1 2 3
    1 3 2
    2 1 3
    2 3 1
    3 1 2
    3 2 1
    ```

    This example shows that the elements are reordered such that each new permutation represents the next bigger value (132 is bigger than 123, 213 is bigger than 132, etc.), using `operator<()` of the data type to which the iterators point. The value `true` is returned if a reordering took place, the value `false` is returned if no reordering took place, which is the case if the sequence represents the last (biggest) value. In that case, the sequence is also sorted using `operator<()`.
  - The second prototype: the next permutation given the sequence of elements in the range `[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 <iterator>
#include <iostream>
#include <string>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(first.c_str(), second.c_str()) < 0;
        }
};

using namespace std;

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()));

    return 0;
}
/*
    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 file:

```
#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 alle elements in the range [nth + 1, last) are greater than the element pointed to by nth. The two subsets themselves are not sorted. The operator<() of the data type to which the iterators point is used to compare the elements.
  - The second prototype: all elements in the range [first, last) are sorted relative to the element pointed to by nth: all elements in the range [left, nth) are smaller than the element pointed to by nth, and alle 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 <iterator>
#include <functional>
using namespace std;

int main()
{
    int ia[] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};

    nth_element(ia, ia + 3, ia + 10);

    cout << "sorting with respect to " << ia[3] << 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;

    return 0;
}
/*
    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 file:

```
#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 data type to which the iterators point. 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>
#include <iterator>
using namespace std;

int main()
{
    int ia[] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};

    partial_sort(ia, ia + 3, ia + 10);

    cout << "find the 3 smallest elements:\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;

    return 0;
}
/*
    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 file:

```
#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 data type to which the iterators point. 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 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>
#include <iterator>
using namespace std;

int main()
{
    int ia[] = {1, 10, 3, 8, 5, 6, 7, 4, 9, 2};
    int ia2[6];

    partial_sort_copy(ia, ia + 10, ia2, ia2 + 6);

    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << 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;
```

```
    return 0;
}
/*
    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 file:

```
#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: each element in the range `[result, <returned OutputIterator>)` receives a value which is obtained by adding the elements in the corresponding range of the range `[first, last)`. The first element in the resulting range will be equal to the element pointed to by `first`.
  - The second prototype: the value of each element in the range `[result, <returned OutputIterator>)` is obtained by applying the binary operator `op` to the previous element in the resulting range and the corresponding element in the range `[first, last)`. The first element in the resulting range will be equal to the element pointed to by `first`.
- Example:

```
#include <numeric>
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int ia[] = {1, 2, 3, 4, 5};
    int ia2[5];

    copy(ia2,
        partial_sum(ia, ia + 5, ia2),
```

```
            ostream_iterator<int>(cout, " "));
        cout << endl;

        copy(ia2,
            partial_sum(ia, ia + 5, ia2, multiplies<int>()),
            ostream_iterator<int>(cout, " "));
        cout << endl;

        return 0;
    }
    /*
        Generated output:

        1 3 6 10 15
        1 2 6 24 120
    */
```

17.4.37 partition()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `BidirectionalIterator partition(BidirectionalIterator first, BidirectionalIterator last, UnaryPredicate pred);`
- Description:
  - All elements in the range `[first, last)` for which the unary predicate `pred` evaluates as `true` are placed before the elements which evaluate as `false`. The return value points just beyond the last element in the partitioned range for which `pred` evaluates as `true`.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class LessThan
{
    int d_x;
    public:
        LessThan(int x)
        :
            d_x(x)
        {}
        bool operator()(int value)
        {
            return value <= d_x;
        }
};
```

```
using namespace std;

int main()
{
    int ia[] = {1, 3, 5, 7, 9, 10, 2, 8, 6, 4};
    int *split;

    split = partition(ia, ia + 10, LessThan(ia[9]));
    cout << "Last element <= 4 is ia[" << split - ia - 1 << "]\n";

    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    Last element <= 4 is ia[3]
    1 3 4 2 9 10 7 8 6 5
*/
```

17.4.38 prev_permutation()

- Header file:

```
#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 such 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 data type to which the iterators point.
  - The second prototype: the previous permutation given the sequence of elements in the range `[first, last)` is determined. 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>
#include <iterator>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return strcasecmp(first.c_str(), second.c_str()) < 0;
        }
};

using namespace std;

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";

    return 0;
}
/*
    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 file:

```
#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. Alternatively, the random number generator may be a function expecting an `int remaining` parameter and returning an `int` randomvalue in the range `[0, remaining)`. Note that when a function object is used, it cannot be an anonymous object. The function in the example uses a procedure outlined in *Press et al.* (1992) **Numerical Recipes in C: The Art of Scientific Computing** (New York: Cambridge University Press, (2nd ed., p. 277)).
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <time.h>
#include <iterator>

int randomValue(int remaining)
{
    return static_cast<int>
            ( ((0.0 + remaining) * rand()) / (RAND_MAX + 1.0) );
}

class RandomGenerator
{
    public:
        RandomGenerator()
        {
            srand(time(0));
        }
        int operator()(int remaining) const
        {
            return randomValue(remaining);
        }
```

```
};


void show(std::string *begin, std::string *end)
{
    std::copy(begin, end,
                    std::ostream_iterator<std::string>(std::cout, " "));
    std::cout << std::endl << std::endl;
}

using namespace std;

int main()
{
    string words[] =
                { "kilo", "lima", "mike", "november", "oscar", "papa"};
    size_t const size = sizeof(words) / sizeof(string);

    cout << "Using Default Shuffle:\n";
    random_shuffle(words, words + size);
    show(words, words + size);

    cout << "Using RandomGenerator:\n";
    RandomGenerator rg;
    random_shuffle(words, words + size, rg);
    show(words, words + size);

    srand(time(0) << 1);
    cout << "Using the randomValue() function:\n";
    random_shuffle(words, words + size, randomValue);
    show(words, words + size);

    return 0;
}
/*
    Generated output (for example):

    Using Default Shuffle:
    lima oscar mike november papa kilo

    Using RandomGenerator:
    kilo lima papa oscar mike november

    Using the randomValue() function:
    mike papa november kilo oscar lima
*/
```

17.4.40 remove()

- Header file:

```
#include <algorithm>
```

- Function prototype:

  - `ForwardIterator remove(ForwardIterator first, ForwardIterator last, Type const &value);`

- Description:
  - The elements in the range pointed to by `[first, last)` are reordered 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 that can be removed after reordering. The range `[returnvalue, last)` is called the *leftover* of the algorithm. Note that the leftover may contain elements different from `value`, but these elements can be removed safely, as such elements will also be present in the range `[first, return value)`. Such duplication is the result of the fact that the algorithm *copies*, rather than *moves* elements into new locations. The function uses `operator==()` of the data type to which the iterators point to determine which elements to remove.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "alpha", "alpha", "papa", "quebec" };
    string *removed;
    size_t const size = sizeof(words) / sizeof(string);

    cout << "Removing all \"alpha\"s:\n";
    removed = remove(words, words + size, "alpha");
    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
         << "Leftover elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    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 const &value);`
- Description:
  - The elements in the range pointed to by `[first, last)` not matching `value` are copied to the range `[result, returnvalue)`, where `returnvalue` is the value returned by the function. The range `[first, last)` is not modified. The function uses `operator==()` of the data type to which the iterators point to determine which elements not to copy.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    string remaining
            [
                size -
                count_if
                (
                    words, words + size,
                    bind2nd(equal_to<string>(), string("alpha"))
                )
            ];
    string *returnvalue =
            remove_copy(words, words + size, remaining, "alpha");

    cout << "Removing all \"alpha\"s:\n";
    copy(remaining, returnvalue, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    Removing all "alpha"s:
    kilo lima mike november oscar papa quebec
*/
```

17.4.42 remove_copy_if()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `OutputIterator remove_copy_if(InputIterator first, InputIterator last, OutputIterator result, UnaryPredicate pred);`
- Description:
  - The elements in the range pointed to by `[first, last)` for which the unary predicate `pred` returns `true` are 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>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    string remaining[
                    size -
                    count_if
                    (
                        words, words + size,
                        bind2nd(equal_to<string>(), "alpha")
                    )
                ];
    string *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;

    return 0;
}
/*
    Generated output:

    Removing all "alpha"s:
    kilo lima mike november oscar papa quebec
*/
```

17.4.43 remove_if()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `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 elements for which the predicate `pred` returns `false`, but these can safely be removed, as such elements will also be present in the range `[first, returnvalue)`. Such duplication is the result of the fact that the algorithm *copies*, rather than *moves* elements into new locations.
- Example:

```
#include <functional>
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);

    cout << "Removing all \"alpha\"s:\n";

    string *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;

    return 0;
}
/*
    Generated output:

    Removing all "alpha"s:
    kilo lima mike november oscar papa quebec
```

```
    Trailing elements are:
    oscar alpha alpha papa quebec
*/
```

17.4.44 replace()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `ForwardIterator replace(ForwardIterator first, ForwardIterator last, Type const &oldvalue, Type const &newvalue);`
- Description:
  - All elements equal to `oldvalue` in the range pointed to by `[first, last)` are replaced by a copy of `newvalue`. The algorithm uses `operator==()` of the data type to which the iterators point.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);

    replace(words, words + size, string("alpha"), string("ALPHA"));
    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA ALPHA papa quebec
*/
```

17.4.45 replace_copy()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `OutputIterator replace_copy(InputIterator first, InputIterator last, OutputIterator result, Type const &oldvalue, Type const &newvalue);`
- Description:
  - All elements equal to `oldvalue` in the range pointed to by `[first, last)` are replaced by a copy of `newvalue` in a new range `[result, returnvalue)`, where `returnvalue` is the return value of the function. The algorithm uses `operator==()` of the data type to which the iterators point.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    string remaining[size];

    copy
    (
        remaining,
        replace_copy(words, words + size, remaining, string("alpha"),
                                                      string("ALPHA")),
        ostream_iterator<string>(cout, " ")
    );
    cout << endl;

    return 0;
}
/*
    Generated output:

    kilo ALPHA lima mike ALPHA november ALPHA oscar ALPHA ALPHA papa quebec
*/
```

17.4.46 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>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november",
          "alpha", "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    string 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;

    return 0;
}
/*
    Generated output (all on one line):

    ALPHA ALPHA ALPHA mike ALPHA november ALPHA oscar ALPHA ALPHA
                                                         papa quebec
*/
```

17.4.47 replace_if()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `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>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);

    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
*/
```

17.4.48 reverse()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `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>
using namespace std;

int main()
{
    string line;

    while (getline(cin, line))
    {
```

```
            reverse(line.begin(), line.end());
            cout << line << endl;
        }

        return 0;
    }
```

17.4.49 reverse_copy()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `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>
using namespace std;

int main()
{
    string line;

    while (getline(cin, line))
    {
        size_t    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;
    }
    return 0;
}
```

17.4.50 rotate()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `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, keeping the order of the elements in the two subsets intact.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa",
          "echo", "foxtrot", "golf", "hotel", "india", "juliet" };
    size_t const size = sizeof(words) / sizeof(string);
    size_t const midsize = 6;

    rotate(words, words + midsize, words + size);

    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    echo foxtrot golf hotel india juliet kilo lima mike november oscar papa
*/
```

17.4.51 rotate_copy()

- Header file:

```
#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. The original order of the elements in the two subsets is not altered.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa",
          "echo", "foxtrot", "golf", "hotel", "india", "juliet" };
    size_t const size = sizeof(words) / sizeof(string);
    size_t midsize = 6;
    string out[size];

    copy(out,
        rotate_copy(words, words + midsize, words + size, out),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    echo foxtrot golf hotel india juliet kilo lima mike november oscar papa
*/
```

17.4.52 search()

- Header file:

```
#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 data type to which the iterators point. If no such location exists, `last1` is returned.
  - The second prototype: an iterator into the first range `[first1, last1)` is returned where the elements in the range `[first2, last2)` are found, using the provided binary predicate `pred` to compare the elements in the two ranges. If no such location exists, `last1` is returned.

- Example:

```
#include <algorithm>
#include <iostream>
#include <iterator>

class absInt
{
    public:
        bool operator()(int i1, int i2)
        {
            return abs(i1) == abs(i2);
        }
};

using namespace std;

int main()
{
    int range1[] =  {-2, -4, -6, -8, 2, 4, 6, 8};
    int 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;

    return 0;
}
/*
    Generated output:

    6 8
    -6 -8 2 4 6 8
*/
```

17.4.53 search_n()

- Header file:

```
#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==()` of the data type to which the iterators point to compare the elements. If no such location exists, `last1` is returned.
  - The second prototype: an iterator into the first range `[first1, last1)` is returned where `n` 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>
#include <iterator>

class absInt
{
    public:
        bool operator()(int i1, int i2)
        {
            return abs(i1) == abs(i2);
        }
};

using namespace std;

int main()
{
    int range1[] =  {-2, -4, -4, -6, -8, 2, 4, 4, 6, 8};
    int 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;

    return 0;
}
/*
```

```
    Generated output:

    4 4
    -4 -4 -6 -8 2 4 4
*/
```

17.4.54 set_difference()

- Header file:

```
#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` returned by the function. The elements in the two ranges must have been sorted using `operator<()` of the data type to which the iterators point.
  - The second prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are not present in the range `[first2, last2)` is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseLess
{
    public:
        bool operator()(std::string const &left, std::string const &right)
        {
            return strcasecmp(left.c_str(), right.c_str()) < 0;
        }
};

using namespace std;

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november",
                      "oscar", "papa", "quebec" };
```

```
        string set2[] = { "papa", "quebec", "romeo"};
        string result[7];
        string *returned;

        copy(result,
            set_difference(set1, set1 + 7, set2, set2 + 3, result),
            ostream_iterator<string>(cout, " "));
        cout << endl;

        string set3[] = { "PAPA", "QUEBEC", "ROMEO"};

        copy(result,
            set_difference(set1, set1 + 7, set3, set3 + 3, result,
            CaseLess()),
            ostream_iterator<string>(cout, " "));
        cout << endl;

        return 0;
    }
    /*
        Generated output:

        kilo lima mike november oscar
        kilo lima mike november oscar
    */
```

17.4.55 set_intersection()

- Header file:

```
#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 range `[first2, last2)` is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using `operator<()` of the data type to which the iterators point.
  - The second prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are also present in the range `[first2, last2)` is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseLess
{
    public:
        bool operator()(std::string const &left, std::string const &right)
        {
            return strcasecmp(left.c_str(), right.c_str()) < 0;
        }
};

using namespace std;

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november",
                      "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo"};
    string result[7];
    string *returned;

    copy(result,
        set_intersection(set1, set1 + 7, set2, set2 + 3, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string set3[] = { "PAPA", "QUEBEC", "ROMEO"};

    copy(result,
        set_intersection(set1, set1 + 7, set3, set3 + 3, result,
        CaseLess()),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    papa quebec
    papa quebec
*/
```

17.4.56 set_symmetric_difference()

- Header file:

```
#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` returned by the function. The elements in the two ranges must have been sorted using `operator<()` of the data type to which the iterators point.
  - The second prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are not present in the range `[first2, last2)` and those in the range `[first2, last2)` that are not present in the range `[first1, last1)` is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseLess
{
    public:
        bool operator()(std::string const &left, std::string const &right)
        {
            return strcasecmp(left.c_str(), right.c_str()) < 0;
        }
};

using namespace std;

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november",
                      "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo"};
    string result[7];
    string *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;

    return 0;
}
/*
    Generated output:

    kilo lima mike november oscar romeo
    kilo lima mike november oscar ROMEO
*/
```

17.4.57 set_union()

- Header file:

  ```
  #include <algorithm>
  ```

- Function prototypes:
  - `OutputIterator set_union(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
  - `OutputIterator set_union(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`
- Description:
  - The first prototype: a sorted sequence of the elements that are present in either the range `[first1, last1)` or the range `[first2, last2)` or in both ranges is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using `operator<()` of the data type to which the iterators point. Note that in the final range each element will appear only once.
  - The second prototype: a sorted sequence of the elements that are present in either the range `[first1, last1)` or the range `[first2, last2)` or in both ranges is returned, starting at `result`, and ending at the `OutputIterator` returned by the function. The elements in the two ranges must have been sorted using `comp` function object. Note that in the final range each element will appear only once.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseLess
```

```
{
    public:
        bool operator()(std::string const &left, std::string const &right)
        {
            return strcasecmp(left.c_str(), right.c_str()) < 0;
        }
};

using namespace std;

int main()
{
    string set1[] = { "kilo", "lima", "mike", "november",
                      "oscar", "papa", "quebec" };
    string set2[] = { "papa", "quebec", "romeo"};
    string result[7];
    string *returned;

    copy(result,
        set_union(set1, set1 + 7, set2, set2 + 3, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string set3[] = { "PAPA", "QUEBEC", "ROMEO"};

    copy(result,
        set_union(set1, set1 + 7, set3, set3 + 3, result,
        CaseLess()),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    kilo lima mike november oscar papa quebec romeo
    kilo lima mike november oscar papa quebec ROMEO
*/
```

17.4.58 sort()

- Header file:

```
#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 data type to which the iterators point.
- The second prototype: the elements in the range `[first, last)` are sorted in ascending order, using the `comp` function object to compare the elements. The binary predicate `comp` should return `true` if its first argument should be placed earlier in the sorted sequence than its second argument.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    string words[] = {"november", "kilo", "mike", "lima",
                      "oscar", "quebec", "papa"};

    sort(words, words + 7);
    copy(words, words + 7, ostream_iterator<string>(cout, " "));
    cout << endl;

    sort(words, words + 7, greater<string>());
    copy(words, words + 7, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    kilo lima mike november oscar papa quebec
    quebec papa oscar november mike lima kilo
*/
```

17.4.59 stable_partition()

- Header file:

```
#include <algorithm>
```

- Function prototype:
  - `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 equal 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>
#include <iterator>
using namespace std;

int main()
{
    int org[] = {1, 3, 5, 7, 9, 10, 2, 8, 6, 4};
    int ia[10];
    int *split;

    copy(org, org + 10, ia);
    split = partition(ia, ia + 10, bind2nd(less_equal<int>(), ia[9]));
    cout << "Last element <= 4 is ia[" << split - ia - 1 << "]\n";

    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << 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;

    return 0;
}
/*
    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 file:

```
#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 data type to which the iterators point: the relative order of equal elements is kept.
  - The second prototype: the elements in the range `[first, last)` are stable-sorted in ascending order, using the `comp` binary predicate to compare the elements. This predicate should return `true` if its first argument should be placed before its second argument in the sorted set of element.
- Example (annotated below):

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <iterator>

typedef std::pair<std::string, std::string>  pss;   // 1 (see the text)

namespace std
{
    ostream &operator<<(ostream &out, pss const &p)                 // 2
    {
        return out << "    " << p.first << " " << p.second << endl;
    }
}

class sortby
{
    std::string pss::*d_field;
    public:
        sortby(std::string pss::*field)                             // 3
        :
            d_field(field)
        {}

        bool operator()(pss const &p1, pss const &p2) const         // 4
        {
            return p1.*d_field < p2.*d_field;
        }
};

using namespace std;

int main()
{
    vector<pss> namecity;                                           // 5

    namecity.push_back(pss("Hampson",   "Godalming"));
    namecity.push_back(pss("Moran",     "Eugene"));
    namecity.push_back(pss("Goldberg",  "Eugene"));
    namecity.push_back(pss("Moran",     "Godalming"));
    namecity.push_back(pss("Goldberg",  "Chicago"));
    namecity.push_back(pss("Hampson",   "Eugene"));
```

```
    sort(namecity.begin(), namecity.end(), sortby(&pss::first));    // 6

    cout << "sorted by names:\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));

    return 0;
}
/*
    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 using multiple hierarchical criteria. The example deserves some additional attention:

1. First, a `typedef` is used to reduce the clutter that occurs from the repeated use of `pair<string, string>`.
2. Next, `operator<<()` is overloaded to be able to insert a `pair` into an `ostream` object. This is merely a service function to make life easy. Note, however, that this function is put in the `std` namespace. If this namespace wrapping is omitted, it won't be used, as `ostream`'s `operator<<()` operators must be part of the `std` namespace.
3. 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>`.
4. 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`.
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 prototype:
  - `void swap(Type &object1, Type &object2);`
- Description:
  - The elements `object1` and `object2` exchange their values.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string first[] = {"alpha", "bravo", "charley"};
    string second[] = {"echo", "foxtrot", "golf"};
    size_t const n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    for (size_t idx = 0; idx < n; ++idx)
        swap(first[idx], second[idx]);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    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 prototype:
  - `ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 result);`
- Description:
  - The elements in the range pointed to by `[first1, last1)` are swapped with the elements in the range `[result, returnvalue)`, where `returnvalue` is the value returned by the function. The two ranges must be disjoint.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>
using namespace std;

int main()
{
    string first[] = {"alpha", "bravo", "charley"};
    string second[] = {"echo", "foxtrot", "golf"};
    size_t const n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << 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;

    return 0;
}
/*
```

```
    Generated output:

    Before:
    alpha bravo charley
    echo foxtrot golf
    After:
    echo foxtrot golf
    alpha bravo charley
*/
```

17.4.63 transform()

- Header file:

```
#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 `[first1, last1)` and the corresponding element in the second range starting at `first2`. The resulting values are stored in the range starting at `result`. The return value points just beyond the last generated element.
- Example:

```
#include <functional>
#include <vector>
#include <algorithm>
#include <iostream>
#include <string>
#include <cctype>
#include <iterator>

class Caps
{
    public:
        std::string operator()(std::string const &src)
        {
            std::string tmp = src;

            transform(tmp.begin(), tmp.end(), tmp.begin(), toupper);
            return tmp;
        }
};
```

```
using namespace std;

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;

    return 0;
}
/*
    Generated output:

    ALPHA BRAVO CHARLEY
    1 4 9 16 25
*/
```

the following differences between the `for_each()` (section 17.4.17) and `transform()` generic algorithms should be noted:

- With `transform()` the *return value* of the function object's `operator()()` member is used; the argument that is passed to the `operator()()` member itself is not changed.
- With `for_each()` the function object's `operator()()` receives a reference to an argument, which itself may be changed by the function object's `operator()()`.

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: using `operator==()`, all but the first of consecutively equal elements of the data type to which the iterators point in the range pointed to by `[first, last)`

are relocated to the end of the range. The returned forward iterator marks the beginning of the *leftover*. All elements in the range `[first, return-value)` are unique, all elements in the range `[return-value, last)` are equal to elements in the range `[first, return-value)`.

- The second prototype: all but the first of consecutive elements in the range pointed to by `[first, last)` for which the binary predicate `pred` (expecting two arguments of the data type to which the iterators point) returns `true`, are relocated to the end of the range. The returned forward iterator marks the beginning of the *leftover*. For all pairs of elements in the range `[first, return-value)` `pred` returns `false` (i.e., are *unique*), while `pred` returns `true` for a combination of, as its first operand, an element in the range `[return-value, last)` and, as its second operand, an element in the range `[first, return-value)`.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <iterator>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return !strcasecmp(first.c_str(), second.c_str());
        }
};

using namespace std;

int main()
{
    string words[] = {"alpha", "alpha", "Alpha", "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);

    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;

    return 0;
}
/*
    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 data type to which the iterators point) are copied only once. The returned output iterator points just beyond the last copied element.
  - The second prototype: the elements in the range `[first, last)` are copied to the resulting container, starting at `result`. Consecutive elements in the range pointed to by `[first, last)` for which the binary predicate `pred` returns `true` are copied only once. The returned output iterator points just beyond the last copied element.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <iterator>

class CaseString
{
    public:
        bool operator()(std::string const &first,
                        std::string const &second) const
        {
            return !strcasecmp(first.c_str(), second.c_str());
        }
};

using namespace std;

int main()
```

```
{
    string words[] = {"oscar", "Alpha", "alpha", "alpha",
                                                "papa", "quebec" };
    size_t const size = sizeof(words) / sizeof(string);
    vector<string> remaining;

    unique_copy(words, words + size, back_inserter(remaining));

    copy(remaining.begin(), remaining.end(),
            ostream_iterator<string>(cout, " "));
    cout << endl;

    vector<string> remaining2;

    unique_copy(words, words + size,
                        back_inserter(remaining2), CaseString());

    copy(remaining2.begin(), remaining2.end(),
            ostream_iterator<string>(cout, " "));
    cout << endl;

    return 0;
}
/*
    Generated output:

    oscar Alpha alpha papa quebec
    oscar Alpha papa quebec
*/
```

17.4.66 upper_bound()

- Header file:

  ```
  #include <algorithm>
  ```

- Function prototypes:
  - `ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, Type const &value);`
  - `ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, Type const &value, Compare comp);`
- Description:
  - The first prototype: the sorted elements stored in the iterator range `[first, last)` are searched for the first element that is greater than `value`. The returned iterator marks the first location in the sequence where `value` can be inserted without breaking the sorted order of the elements, using `operator<()` of the data type to which the iterators point. If no such element is found, `last` is returned.
  - The second prototype: the elements implied by the iterator range `[first, last)` must have been sorted using the `comp` function or function object. Each element in the range is compared to `value` using the `comp` function. An iterator to the first element for which the binary predicate `comp`, applied to the elements of the range and `value`, returns `true` is returned. If no such element is found, `last` is returned.

- Example:

```
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>
using namespace std;

int main()
{
    int         ia[] = {10, 15, 15, 20, 30};
    size_t    n = sizeof(ia) / sizeof(int);

    cout << "Sequence: ";
    copy(ia, ia + n, ostream_iterator<int>(cout, " "));
    cout << 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;

    return 0;
}
/*
    Generated output:

    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 kind of binary tree which can be represented by an array. In the standard heap, the key of an element is not smaller than the key of its children. This kind of heap is called a *max heap*. A tree in which numbers are keys could be organized as shown in figure 17.1. Such a tree may also be

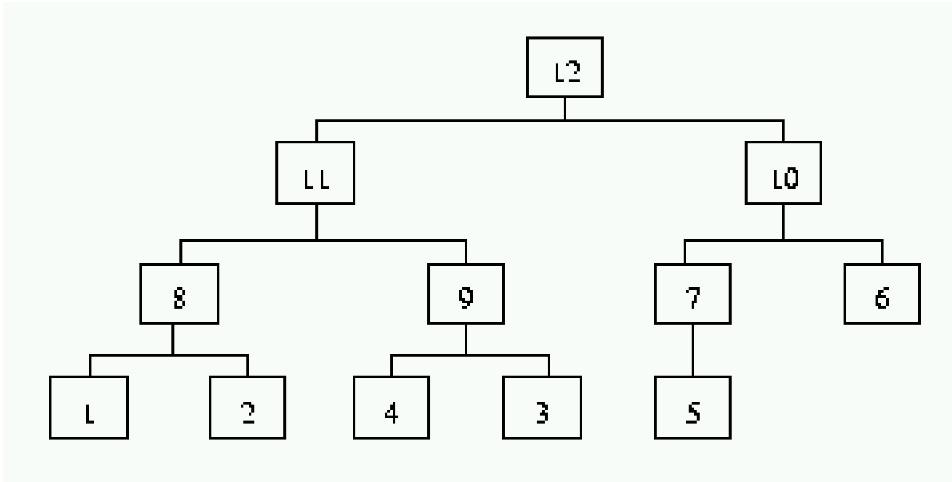

Figure 17.1: A binary tree representation of a heap

organized in an array:

```
12, 11, 10, 8, 9, 7, 6, 1, 2, 4, 3, 5
```

In the following description, keep two pointers into this array in mind: a pointer `node` indicates the location of the next node of the tree, a pointer `child` points to the next element which is a child of the `node` pointer. Initially, `node` points to the first element, and `child` points to the second element.

- `*node++ (== 12)`. 12 is the top node. its children are `*child++` (11) and `*child++` (10), both less than 12.
- The next node (`*node++ (== 11)`), in turn, has `*child++` (8) and `*child++` (9) as its children.
- The next node (`*node++ (== 10)`) has `*child++` (7) and `*child++` (6) as its children.
- The next node (`*node++ (== 8)`) has `*child++` (1) and `*child++` (2) as its children.
- Then, node (`*node++ (== 9)`) has children `*child++` (4) and `*child++` (3).
- Finally (as far as children are concerned) (`*node++ (== 7)`) has one child `*child++` (5)

Since `child` now points beyond the array, the remaining nodes have no children. So, nodes 6, 1, 2, 4, 3 and 5 don't have children.

Note that the left and right branches are not ordered: 8 is less than 9, but 7 is larger than 6.

The heap is created by traversing a binary tree level-wise, starting from the top node. The top node is 12, at the zeroth level. At the first level we find 11 and 10. At the second level 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 show the prototypes of the heap-algorithms, the final subsection provides a small example in which the heap algorithms are used.

17.4.67.1 The 'make_heap()' function

- Header file:

```
#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 data type to which the iterators point.
  - The second prototype: the elements in the range `[first, last)` are reordered to form a max-heap, using the binary comparison function object `comp` to compare elements.

17.4.67.2 The 'pop_heap()' function

- Header file:

```
#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 data type to which the iterators point.
  - 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 max-heap, using the binary comparison function object `comp` to compare elements.

17.4.67.3 The 'push_heap()' function

- Header file:

```
#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 data type to which the iterators point.
  - 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 max-heap, using the binary comparison function object comp to compare elements.

17.4.67.4 The 'sort_heap()' function

- Header file:

```
#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 data type to which the iterators point.
  - The second prototype: assuming the elements in the range [first, last) form a valid heap, the elements in the range [first, last) are sorted, using the binary comparison function object comp to compare elements.

17.4.67.5 An example using the heap functions

Here is an example showing the various generic algorithms manipulating heaps:

```
#include <algorithm>
#include <iostream>
#include <functional>
#include <iterator>

void show(int *ia, char const *header)
{
    std::cout << header << ":\n";
    std::copy(ia, ia + 20, std::ostream_iterator<int>(std::cout, " "));
    std::cout << std::endl;
}

using namespace std;

int main()
{
```

```
    int ia[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                11, 12, 13, 14, 15, 16, 17, 18, 19, 20};

    make_heap(ia, ia + 20);
    show(ia, "The values 1-20 in a max-heap");

    pop_heap(ia, ia + 20);
    show(ia, "Removing the first element (now at the end)");

    push_heap(ia, ia + 20);
    show(ia, "Adding 20 (at the end) to the heap again");

    sort_heap(ia, ia + 20);
    show(ia, "Sorting the elements in the heap");


    make_heap(ia, ia + 20, greater<int>());
    show(ia, "The values 1-20 in a heap, using > (and beyond too)");

    pop_heap(ia, ia + 20, greater<int>());
    show(ia, "Removing the first element (now at the end)");

    push_heap(ia, ia + 20, greater<int>());
    show(ia, "Re-adding the removed element");

    sort_heap(ia, ia + 20, greater<int>());
    show(ia, "Sorting the elements in the heap");

    return 0;
}
/*
    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
    Re-adding the removed element:
    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

Template functions

C++ supports syntactical constructs allowing programmers to define and use completely general (or abstract) functions or classes, based on generic types and/or (possibly inferred) constant values. In the chapters on abstract containers (chapter 12) and the `STL` (chapter 17) we've already used these constructs, commonly known as the *template mechanism*.

The template mechanism allows us to specify classes and algorithms, fairly independently of the actual types for which the templates will eventually be used. Whenever the template is used, the compiler will generate code, tailored to the particular data type(s) used with the template. This code is generated compile-time from the template's definition. The piece of generated code is called an *instantiation* of the template.

In this chapter the syntactical peculiarities of templates will be covered. The notions of *template type parameter*, *template non-type parameter*, and *template function* will be introduced, and several examples of templates will be offered, both in this chapter and in chapter 20, providing concrete examples of **C++**. Template *classes* are covered in chapter 19.

Templates offered standard by the language already cover containers allowing us to construct both highly complex and standard data structures commonly used in computer science. Furthermore, the `string` (chapter 4) and stream (chapter 5) classes are commonly implemented using templates. So, templates play a central role in present-day **C++**, and should absolutely not be considered an esoteric feature of the language.

Templates should be approached somewhat similarly as generic algorithms: they're a *way of life*; a **C++** software engineer should actively look for opportunities to use them. Initially, templates appear to be rather complex, and you might be tempted to turn your back on them. However, in time their strengths and benefits will be more and more appreciated. Eventually you'll be able to recognize opportunities for using templates. That's the time where your efforts should no longer focus on constructing concrete (i.e., non-template) functions or classes, but on constructing templates.

This chapter starts by introducing *template functions*. The emphasis is on the required syntax when defining such functions. This chapter lays the foundation upon which the next chapter, introducing template classes and offering several real-life examples, is built.

18.1 Defining template functions

A template function's definition is very similar to the definition of a normal function. A template function has a function head, a function body, a return type, possibly overloaded definitions, etc..

However, different from concrete functions, template functions always use one or more *formal types*: types for which almost any exising (class or primitive) type could be used. Let's start with a simple example. The following function `add()` expects two arguments, and returns their sum:

```
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}
```

Note how closely the above function's definition follows its description: it gets two arguments, and returns its sum. Now consider what would happen if we would have to define this function for, e.g., `int` values. We would have to define:

```
int add(int const &lvalue, int const &rvalue)
{
    return lvalue + rvalue;
}
```

So far, so good. However, were we to add to doubles, we would have to overload this function so that its overloaded version accepts doubles:

```
double add(double const &lvalue, double const &rvalue)
{
    return lvalue + rvalue;
}
```

There is no end to the number of overloaded versions we might be forced to construct: an overloaded version for `std::string`, for `size_t`, for .... In general, we would need an overloaded version for every type supporting `operator+()` and a copy constructor. All these overloaded versions of basically the same function are required because of the strongly typed nature of **C++**. Because of this, a truly generic function cannot be constructed without resorting to the template mechanism.

Fortunately, we've already seen the meat and bones of a template function. Our initial function `add()` actually is an implementation of such a function. However, it isn't a full template definition yet. If we would give the first `add()` function to the compiler, it would produce an error message like:

```
error: 'Type' was not declared in this scope
error: parse error before 'const'
```

And rightly so, as we failed to define `Type`. The error is prevented when we change `add()` into a full template definition. To do this, we look at the function's implementation and decide that `Type` is actually a *formal* typename. Comparing it to the alternate implementations, it will be clear that we could have changed `Type` into `int` to get the first implementation, and into `double` to get the second.

The full template definition allows for this formal character of the `Type` typename. Using the keyword `template`, we prefix one line to our initial definition, obtaining the following template function definition:

```
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue)
```

```
{
    return lvalue + rvalue;
}
```

In this definition we distinguish:

- The keyword `template`, starting a template definition or declaration.
- The angle bracket enclosed list following `template`: it is a list, containing one or more comma-separated elements. This angle bracket enclosed list is called the *template parameter list*. When multiple elements are used, it could look like, e.g.,

  ```
  typename Type1, typename Type2
  ```

- Inside the template parameter list we find the *formal type name* `Type`. It is a formal type name, comparable to a formal parameter name in a function's definition. Up to now we've only encountered formal variable names with functions. The *types* of the parameters were always known by the time the function was defined. Templates escalate the notion of formal names one step further up the ladder, allowing type names to be formalized, rather than just the formal parameter variable names themselves. The fact that `Type` is a formal type name is indicated by the keyword `typename`, prefixed to `Type` in the template parameter list. A formal type name like `Type` is also called a *template type parameter*. Template non-type parameters also exist, and are introduced below.

  Other texts on **C++** sometimes use the keyword `class` where we use `typename`. So, in other texts template definitions might start with a line like:

  ```
  template <class Type>
  ```

  Using `class` instead of `typename` is now, however, considered an anachronism, and is deprecated: a template type parameter is, after all, a type name.

- The function head: it is like a normal function head, albeit that the template's type parameters must be used in its parameter list. When the function is actually called, using actual arguments having actual types, these actual types are then used by the compiler to determine which version (overloaded to fit the actual argument types) of the template function must be used. At this point (i.e., where the function is called), the compiler will create the concrete function, a process called *instantiation*. The function head may also use a formal type to specify its return value. This feature was actually used in the `add()` template's definition.
- The function parameters are specified as `Type const &` parameters. This has the usual meaning: the parameters are references to `Type` objects or values that will not be modified by the function.
- The function body: it is like a normal function body. In the body the formal type names may be used to define or declare variables, which may then be used as any other local variable. Even so, there are some restrictions. Looking at `add()`'s body, it is clear that `operator+()` is used, as well as a copy constructor, as the function returns a value. This allows us to formulate the following restrictions for the formal type `Type`:
  - `Type` should support `operator+()`
  - `Type` should support a copy constructor

  Consequently, while `Type` could be a `std::string`, it could never be an `ostream`, as neither `operator+()` nor the copy constructor are available for streams.

Normal scope rules and identifier visibility rules apply to template definitions. Formal typenames overrule, within the template definition's scope, any identifiers having identical names having wider scopes.

Look again at the function's parameters, as defined in its parameter list. By specifying `Type const &` rather than `Type` superfluous copying is prevented, at the same time allowing values of primitive types to be passed as arguments to the function. So, when `add(3, 4)` is called, `int(4)` will be assigned to `Type const &rvalue`. In general, function parameters should be defined as `Type const &` to prevent unnecessary copying. The compiler is smart enough to handle 'references to references' in this case, which is something the language normally does not supports. For example, consider the following `main()` function (here and in the following simple examples assuming the template and required headers and namespace declarations have been provided):

```
int main()
{
    size_t const &uc = size_t(4);
    cout << add(uc, uc) << endl;
}
```

Here `uc` is a reference to a constant `size_t`. It is passed as argument to `add()`, thereby initializing `lvalue` and `rvalue` as `Type const &` to `size_t const &` values, with the compiler interpreting `Type` as `size_t`. Alternatively, the parameters might have been specified using `Type &`, rather than `Type const &`. The disadvantage of this (non-const) specification being that temporary values cannot be passed to the function anymore. The following will fail to compile:

```
int main()
{
    cout << add(string("a"), string("b")) << endl;
}
```

Here, a `string const &` cannot be used to initialize a `string &`. On the other hand, the following *will* compile, with the compiler deciding that `Type` should be considered a `string const`:

```
int main()
{
    string const &s = string("a");
    cout << add(s, s) << endl;
}
```

What can we deduce from these examples?

- In general, function parameters should be specified as `Type const &` parameters to prevent unnecessary copying.
- The template mechanism is fairly flexible, in that it will interpret formal types as plain types, const types, pointer types, etc., depending on the actually provided types. The rule of thumb is that the formal type is used as a generic mask for the actual type, with the formal type name covering whatever part of the actual type must be covered. Some examples, assuming the parameter is defined as `Type const &`:

argument type	**Type ==**
`size_t const`	`size_t`
`size_t`	`size_t`
`size_t *`	`size_t *`
`size_t const *`	`size_t const *`

As a second example of a template function, consider the following function definition:

```
template <typename Type, size_t Size>
Type sum(Type const (&array)[Size])
{
    Type t = Type();

    for (size_t idx = 0; idx < Size; idx++)
        t += array[idx];

    return t;
}
```

This template definition introduces the following new concepts and features:

- Its template parameter list has two elements. Its first element is a well-known template type parameter, but its second element has a very specific type: an `size_t`. Template parameters of specific (i.e., non-formal) types used in template parameter lists are called *template non-type parameters*. A template *non-type parameter* 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 `size_t`.
- Looking at the function's head, we see one parameter:

  ```
  Type const (&array)[Size]
  ```

  This parameter defines `array` as a reference parameter to an array having `Size` elements of type `Type`, that may not be modified.
- In the parameter definition, both `Type` and `Size` are used. `Type` is of course the template's type parameter `Type`, but `Size` is also a template parameter. It is an `size_t`, whose value must be inferable by the compiler when it compiles an actual call of the `sum()` template function. Consequently, `Size` must be a `const` value. Such a constant expression is called a *template non-type parameter*, and it is named in the template's parameter list.
- When the template function is called, the compiler must be able to infer not only `Type`'s concrete value, but also `Size`'s value. Since the function `sum()` only has one parameter, the compiler is only able to infer `Size`'s value from the function's actual argument. It can do so if the provided argument is an array (of known and fixed size), rather than a pointer to `Type` elements. So, in the following `main()` function the first statement will compile correctly, whereas the second statement won't:

  ```
  int main()
  {
      int values[5];
      int *ip = values;

      cout << sum(values) << endl;    // compiles ok
      cout << sum(ip) << endl;        // won't compile
  }
  ```

- Inside the function, the statement `Type t = Type()` is used to initialize `t` to a default value. Note here that no fixed value (like 0) is used. Any type's default value may be obtained using its default constructor, rather than using a fixed numerical value. Of course, not every class accepts a numerical value as an argument to one of its constructors. But all types, even the primitive types, support default constructors (actually, some classes do not implement a default constructor, but most do). The default constructor of primitive types will initialize their variables to 0 (or `false`). Furthermore, the statement `Type t = Type()` is a true initialization: `t` is initialized by `Type`'s default constructor, rather than using `Type`'s copy constructor to assign `Type()`'s copy to `t`. Alternatively, the syntactical construction `Type t(Type())` could have been used.

- Comparable to the first template function, `sum()` also assumes the existence of certain public members in `Type`'s class. This time `operator+=()` and `Type`'s copy constructor.

Like class definitions, template definitions should not contain `using` directives or declarations: the template might be used in a situation where such a directive overrides the programmer's intentions: ambiguities or other conflicts may result from the template's author and the programmer using different `using` directives (E.g, a `cout` variable defined in the `std` namespace and in the programmer's own namespace). Instead, within template definitions only fully qualified names, including all required namespace specifications should be used.

18.2 Argument deduction

In this section we'll concentrate on the process by which the compiler deduces the actual types of the template type parameters when a template function is called, a process called *template parameter deduction*. As we've already seen, the compiler is able to substitute a wide range of actual types for a single formal template type parameter. Even so, not every thinkable conversion is possible. In particular when a function has multiple parameters of the same template type parameter, the compiler is very restrictive in what argument types it will actually accept.

When the compiler deduces the actual types for template type parameters, it will only consider the types of the arguments. Neither local variables nor the function's return value is considered in this process. This is understandable: when a function is called, the compiler will only see the template function's arguments with certainty. At the point of the call it will definitely not see the types of the function's local variables, and the function's return value might not actually be used, or may be assigned to a variable of a subrange (or super-range) type of a deduced template type parameter. So, in the following example, the compiler won't ever be able to call `fun()`, as it has no way to deduce the actual type for the `Type` template type parameter.

```
template <typename Type>
Type fun()              // can never be called
{
    return Type();
}
```

In general, when a function has multiple parameters of identical template type parameters, the actual types must be exactly the same. So, whereas

```
    void binarg(double x, double y);
```

may be called using an `int` and a `double`, with the `int` argument implicitly being converted to a `double`, the corresponding template function cannot be called using an `int` and `double` argument: the compiler won't itself promote `int` to `double` and to decide next that `Type` should be `double`:

```
template <typename Type>
```

```
void binarg(Type const &p1, Type const &p2)
{}

int main()
{
    binarg(4, 4.5); // ?? won't compile: different actual types
}
```

What, then, are the transformations the compiler will apply when deducing the actual types of template type parameters? It will perform only three types of parameter type transformations (and a fourth one to function parameters of any fixed type (i.e., of a non-template function parameter type)). If it cannot deduce the actual types using these transformations, the template function will not be considered. These transformations are:

- *lvalue transformations*, creating an *rvalue* from an *lvalue*;
- *qualification transformations*, inserting a `const` modifier to a non-constant argument type;
- *transformation to a base class instantiated from a class template*, using a template base class when an argument of a template derived class type was provided in the call.
- Standard transformations for template non-type function parameters. This isn't a template parameter type transformation, but it refers to any remaining template non-type parameter of template functions. For these function parameters the compiler will perform any standard conversion it has available (e.g., `int` to `size_t`, `int` to `double`, etc.).

The first three types of transformations will now be discussed and illustrated.

18.2.1 Lvalue transformations

There are three types of *lvalue transformations*:

- **lvalue-to-rvalue transformations.**

  An lvalue-to-rvalue transformation is applied when an `rvalue` is required, and an `lvalue` is used as argument. This happens when a variable is used as argument to a function specifying a *value parameter*. For example,

  ```
  template<typename Type>
  Type negate(Type value)
  {
      return -value;
  }
  int main()
  {
      int x = 5;
      x = negate(x);  // lvalue (x) to rvalue (copies x)
  }
  ```

- **array-to-pointer transformations.**

  An array-to-pointer transformation is applied when the name of an array is assigned to a pointer variable. This is frequently seen with functions defining pointer parameters. When calling such functions, arrays are often specified as their arguments.

The array's address is then assigned to the pointer-parameter, and its type is used to deduce the corresponding template parameter's type. For example:

```
template<typename Type>
Type sum(Type *tp, size_t n)
{
    return accumulate(tp, tp + n, Type());
}
int main()
{
    int x[10];
    sum(x, 10);
}
```

In this example, the location of the array `x` is passed to `sum()`, expecting a pointer to some type. Using the array-to-pointer transformation, `x`'s address is considered a pointer value which is assigned to `tp`, deducing that `Type` is `int` in the process.

- **function-to-pointer transformations.**

  This transformation is most often seen with template functions defining a parameter which is a pointer to a function. When calling such a function the name of a function may be specified as its argument. The address of the function is then assigned to the pointer-parameter, deducing the template type parameter in the process. This is called a function-to-pointer transformation. For example:

```
#include <cmath>

template<typename Type>
void call(Type (*fp)(Type), Type const &value)
{
    (*fp)(value);
}
int main()
{
    call(&sqrt, 2.0);
}
```

  In this example, the address of the `sqrt()` function is passed to `call()`, expecting a pointer to a function returning a `Type` and expecting a `Type` for its argument. Using the function-to-pointer transformation, `sqrt`'s address is considered a pointer value which is assigned to `fp`, deducing that `Type` is `double` in the process. Note that the argument `2.0` could not have been specified as `2`, as there is no `int sqrt(int)` prototype. Also note that the function's first parameter specifies `Type (*fp)(Type)`, rather than `Type (*fp)(Type const &)` as might have been expected from our previous discussion about how to specify the types of template function's parameters, preferring references over values. However, `fp`'s argument `Type` is not a template function parameter, but a parameter of the function `fp` points to. Since `sqrt()` has prototype `double sqrt(double)`, rather than `double sqrt(double const &)`, `call()`'s parameter `fp` *must* be specified as `Type (*fp)(Type)`. It's that strict.

18.2.2 Qualification transformations

A *qualification transformation* adds `const` or `volatile` qualifications to *pointers*. This transformation is applied when the template function's parameter is explicitly defined using a `const` (or `volatile`) modifier, and the function's argument isn't a `const` or `volatile` entity. In that case, the transformation adds `const` or `volatile`, and subsequently deduces the template's type parameter. For example:

```
template<typename Type>
Type negate(Type const &value)
{
    return -value;
}
int main()
{
    int x = 5;
    x = negate(x);
}
```

Here we see the template function's `Type const &value` parameter: a reference to a `const Type`. However, the argument isn't a `const int`, but an `int` that can be modified. Applying a qualification transformation, the compiler adds `const` to `x`'s type, and so it matches `int const x` with `Type const &value`, deducing that `Type` must be `int`.

18.2.3 Transformation to a base class

Although the *construction* of template classes will only be constructed in chapter 19, template classes have already extensively been *used* earlier. For example, abstract containers (covered in chapter 12) are actually defined as template classes. Like concrete classes (i.e., non-template classes), template classes can participate in the construction of class hierarchies. In section 19.9 it is shown how a template class can be derived from another template class.

As template class derivation remains to be covered, the following discussion is necessarily somewhat abstract. Optionally, the reader may of course skip briefly to section 19.9, to read this section thereafter.

In this section it should now be assumed, for the sake of argument, that a template class `Vector` has somehow been derived from a `std::vector`. Furthermore, assume that the following template function has been constructed to sort a vector using some function object `obj`:

```
template <typename Type, typename Object>
void sortVector(std::vector<Type> vect, Object const &obj)
{
    sort(vect.begin(), vect.end(), obj);
}
```

To sort `std::vector<string>` objects case-insensitively, the `class Caseless` could be constructed as follows:

```
class CaseLess
{
    public:
        bool operator()(std::string const &before,
                        std::string const &after) const
        {
            return strcasecmp(before.c_str(), after.c_str()) < 0;
        }
};
```

Now various vectors may be sorted, using `sortVector()`:

```
int main()
{
    std::vector<string> vs;
    std::vector<int> vi;

    sortVector(vs, CaseLess());
    sortVector(vi, less<int>());
}
```

Applying the transformation *transformation to a base class instantiated from a class template*, the template function `sortVectors()` may now also be used to sort `Vector` objects. For example:

```
int main()
{
    Vector<string> vs;      // note: not ‘std::vector’
    Vector<int> vi;

    sortVector(vs, CaseLess());
    sortVector(vi, less<int>());
}
```

In this example, `Vectors` were passed as argument to `sortVector()`. Applying the transformation to a base class instantiated from a class template, the compiler will consider `Vector` to be a `std::vector`, and is thus able to deduce the template's type parameter. A `std::string` for the `Vector vs`, an `int` for `Vector vi`.

Please realize the purpose of the various template parameter type deduction transformations. They do not aim at matching function arguments to function parameters, but having matched arguments to parameters, the transformations may be applied to determine the actual types of the various template type parameters.

18.2.4 The template parameter deduction algorithm

The compiler uses the following algorithm to deduce the actual types of its template type parameters:

- In turn, the template function's parameters are identified using the arguments of the called function.
- For each template parameter used in the template function's parameter list, the template type parameter is matched with the corresponding argument's type (e.g., `Type` is `int` if the argument is `int x`, and the function's parameter is `Type &value`).
- While matching the argument types to the template type parameters, the three allowed transformations (see section 18.2) for template type parameters are applied where necessary.
- If identical template type parameters are used with multiple function parameters, the deduced template types must be exactly the same. So, the next template function cannot be called with an `int` and a `double` argument:

  ```
  template <typename Type>
  Type add(Type const &lvalue, Type const &rvalue)
  ```

```
    {
        return lvalue + rvalue;
    }
```

When calling this template function, two identical types must be used (albeit that the three standard transformations are of course allowed). If the template deduction mechanism does not come up with identical actual types for identical template types, then the template function will not be instantiated.

18.3 Declaring template functions

Up to now, we've only defined template functions. There are various consequences of including template function definitions in multiple source files, none of them serious, but worth knowing.

- Like class interfaces, template definitions are usually included in header files. Every time a header file containing a template definition is read by the compiler, the compiler must process the definition in full, even though it might not actually need the template. This will relatively slow-down the compilation. For example, compiling a template header file like `algorithm` on my old laptop takes about four times the amount of time it takes to compile a plain header file like `cmath`. The header file `iostream` is even harder to process, requiring almost 15 times the amount of time it takes to process `cmath`. Clearly, processing templates is serious business for the compiler.

- Every time a template function is instantiated, its code appears in the resulting object module. However, if multiple instantiations of a template, using the same actual types for its template parameter exist in multiple object files, then the linker will weed out superfluous instantiations. In the final program only one instantiation for a particular set of actual template type parameters will be used (see also section 18.4 for an illustration). Therefore, the linker will have an additional task to perform (*viz.* weeding out multiple instantiations), which will slow down the linking process.

- Sometimes the definitions themselves are not required, but only references or pointers to the templates are required. Requiring the compiler to process the full template definitions in those cases will unnecessarily slow down the compilation process.

Instead of including template definitions again and again in various source files, templates may also be declared. When templates are declared, the compiler will not have to process the template's definitions again and again, and no instantiations will be created on the basis of template declarations alone. Any actually required instantiation must, as holding true for declarations in general, be available elsewhere. Unlike the situation we encounter with concrete functions, which are usually stored in libraries, it is currently not possible to store templates in libraries (although precompiled header files may be implemented in various compilers). Consequently, using template declarations puts a burden on the shoulders of the software engineer, who has to make sure that the required instantiations exist. Below a simple way to accomplish that is introduced.

A template function declaration is simply created: the function's body is replaced by a semicolon. Note that this is exactly identical to the way concrete function declarations are constructed. So, the previously defined template function `add()` can simply be declared as

```
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue);
```

Actually, we've already encountered template declarations. The header file `iosfwd` may be included in sources not requiring instantiations of elements from the class `ios` and its derived classes. For example, in order to compile the *declaration*

```
std::string getCsvline(std::istream &in, char const *delim);
```

it is not necessary to include the `string` and `istream` header files. Rather, a single

```
#include <iosfwd>
```

is sufficient, requiring about one-ninth the amount of time it takes to compile the declaration when `string` and `istream` are included.

18.3.1 Instantiation declarations

So, if declaring template functions speeds up the compilation and the linking phases of a program, how can we make sure that the required instantiations of the template functions will be available when the program is eventually linked together?

For this a variant of a declaration is available, a so-called *explicit instantiation declaration*. An explicit instantiation declaration contains the following elements:

- It starts with the keyword `template`, omitting the template parameter list.
- Next the function's return type and name are specified.
- The function name is followed by a *type specification list*, a list of types between angle brackets, each type specifying the actual type of the corresponding template type parameter in the template's parameter list.
- Finally the function's parameter list is specified, terminated by a semicolon.

Although this is a declaration, it is actually understood by the compiler as a request to instantiate that particular variant of the function.

Using explicit instantiation declarations all instantiations of template functions required by a program can be collected in one file. This file, which should be a normal *source* file, should include the template definition header file, and should next specify the required instantiation declarations. Since it's a source file, it will not be included by other sources. So namespace `using` directives and declarations may safely be used once the required headers have been included. Here is an example showing the required instantiations for our earlier `add()` template, instantiated for `double`, `int`, and `std::string` types:

```
#include "add.h"
#include <string>
using namespace std;

template int add<int>(int const &lvalue, int const &rvalue);
template double add<double>(double const &lvalue, double const &rvalue);
template string add<string>(string const &lvalue, string const &rvalue);
```

If we're sloppy and forget to mention an instantiation required by our program, then the repair can easily be made: just add the missing instantiation declaration to the above list. After recompiling the file and relinking the program we're done.

18.4 Instantiating template functions

A template is not instantiated when its definition is read by the compiler. A template is merely a *recipe* telling the compiler how to create particular code once it's time to do so. It's very much like a recipe in a cooking book: you reading a cake's recipe doesn't mean you have actually cooked that cake by the time you've read the recipe.

So, when is a template function actually instantiated? There are two situations in which the compiler will decide to instantiate templates:

- They are instantiated when they're actually used (e.g., the function `add()` is called with a pair of `size_t` values);
- When addresses of template functions are taken they are instantiated. For example:

```
#include "add.h"

char (*addptr)(char const &, char const &) = add;
```

The location of statements causing the compiler to instantiate a template is called the template's *point of instantiation*. The point of instantiation has serious implications for the template function's code. These implications are discussed in section 18.9.

The compiler is not always able to deduce the template's type parameters unambiguously. In that case the compiler reports an ambiguity which must be solved by the software engineer. Consider the following code:

```
#include <iostream>
#include "add.h"

size_t fun(int (*f)(int *p, size_t n));
double fun(double (*f)(double *p, size_t n));

int main()
{
    std::cout << fun(add) << std::endl;
}
```

When this small program is compiled, the compiler reports an ambiguity it cannot resolve. It has two candidate functions, as for each overloaded version of `fun()` a proper instantiation of `add()` can be constructed:

```
error: call of overloaded 'fun(<unknown type>)' is ambiguous
note: candidates are: int fun(size_t (*)(int*, size_t))
note:                 double fun(double (*)(double*, size_t))
```

Situations like these should of course be avoided. Template functions can only be instantiated if there's no ambiguity. Ambiguities arise when multiple functions emerge from the compiler's function selection mechanism (see section 18.8). It is up to us to resolve these ambiguities. Ambiguities like the above can be resolved using a blunt `static_cast` (as we select among alternatives, all of them possible and available):

```
#include <iostream>
```

```
#include "add.h"

int fun(int (*f)(int const &lvalue, int const &rvalue));
double fun(double (*f)(double const &lvalue, double const &rvalue));

int main()
{
    std::cout << fun(
                    static_cast<int (*)(int const &, int const &)>(add)
                ) << std::endl;
    return 0;
}
```

But if possible, type casts should be avoided. How to avoid casts in situations like these is explained in the next section (18.5).

As mentioned in section 18.3, the linker will remove identical instantiations of a template from the final program, leaving only one instantiation for each unique set of actual template type parameters. Let's have a look at an example showing this behavior of the linker. To illustrate the linker's behavior, we will do as follows:

- First we construct several source files:
  - `source1.cc` defines a function `fun()`, instantiating `add()` for `int`-type arguments, including `add()`'s template definition. It displays `add()`'s address. Here is `source1.cc`:

    ```
    union PointerUnion
    {
        int (*fp)(int const &, int const &);
        void *vp;
    };

    #include <iostream>
    #include "add.h"
    #include "pointerunion.h"

    void fun()
    {
        PointerUnion pu = { add };

        std::cout << pu.vp << std::endl;
    }
    ```

  - `source2.cc` defines the same function, but only declares the proper `add()` template, using a template declaration (*not* an instantiation declaration). Here is `source2.cc`:

    ```
    #include <iostream>
    #include "pointerunion.h"

    template<typename Type>
    Type add(Type const &, Type const &);

    void fun()
    {
        PointerUnion pu = { add };
    ```

```
        std::cout << pu.vp << std::endl;
    }
```

- main.cc again includes add()'s template definition, declares the function fun() and defines main(), defining add() for int-type arguments as well and displaying add()'s function address. It also calls the function fun(). Here is main.cc:

```
#include <iostream>
#include "add.h"
#include "pointerunion.h"

void fun();

int main()
{
    PointerUnion pu = { add };

    fun();
    std::cout << pu.vp << std::endl;
}
```

- All sources are compiled to object modules. Note the different sizes of source1.o (2112 bytes, using g++ version 4.0.4. All sizes reported here may differ somewhat for different compilers and/or run-time libraries) and source2.o (1928 bytes). Since source1.o contains the instantiation of add(), it is somewhat larger than source2.o, containing only the template's declaration. Now we're ready to start our little experiment.

- Linking main.o and source1.o, we obviously link together two object modules, each containing its own instantiation of the same template function. The resulting program produces the following output:

```
0x80486d8
0x80486d8
```

  Furthermore, the size of the resulting program is 9152 bytes.

- Linking main.o and source2.o, we now link together an object module containing the instantiation of the add() template, and another object module containing the mere declaration of the same template function. So, the resulting program cannot but contain a single instantiation of the required template function. This program has exactly the same size, and produces exactly the same output as the first program.

So, from our little experiment we can conclude that the linker will indeed remove identical template instantiations from a final program, and that using mere template declarations will not result in template instantiations.

18.5 Using explicit template types

In the previous section (section 18.4) we've seen that the compiler may encounter ambiguities when attempting to instantiate a template. We've seen an example in which overloaded versions of a function fun() existed, expecting different types of arguments, both of which could have been provided by an instantiation of a template function. The intuitive way to solve such an ambiguity is to use a static_cast type cast, but as noted: if possible, casts should be avoided.

When template functions are involved, such a `static_cast` may indeed neatly be avoided, using *explicit template type arguments*. When explicit template type arguments are used the compiler is explicitly informed about the actual template type parameters it should use when instantiating a template. Here, the function's name is followed by an *actual template parameter type list* which may again be followed by the function's argument list, if required. The actual types mentioned in the actual template parameter list are used by the compiler to 'deduce' the actual types of the corresponding template types of the function's template parameter type list. Here is the same example as given in the previous section, now using explicit template type arguments:

```
#include <iostream>
#include "add.h"

int fun(int (*f)(int const &lvalue, int const &rvalue));
double fun(double (*f)(double const &lvalue, double const &rvalue));

int main()
{
    std::cout << fun(add<int>) << std::endl;
    return 0;
}
```

18.6 Overloading template functions

Let's once again look at our `add()` template. That template was designed to return the sum of two entities. If we would want to compute the sum of three entities, we could write:

```
int main()
{
    add(2, add(3, 4));
}
```

This is a perfectly acceptable solution for the occasional situation. However, if we would have to add three entities regularly, an *overloaded* version of the `add()` function, expecting three arguments, might be a useful thing to have. The solution for this problems is simple: template functions may be overloaded.

To define an overloaded version, merely put multiple definitions of the template in its definition header file. So, with the `add()` function this would be something like:

```
template <typename Type>
Type add(Type const &lvalue, Type const &rvalue)
{
    return lvalue + rvalue;
}

template <typename Type>
Type add(Type const &lvalue, Type const &mvalue, Type const &rvalue)
{
    return lvalue + mvalue + rvalue;
}
```

The overloaded function does not have to be defined in terms of simple values. Like all overloaded functions, just a unique set of function parameters is enough to define an overloaded version. For example, here's an overloaded version that can be used to compute the sum of the elements of a vector:

```
template <typename Type>
Type add(std::vector<Type> const &vect)
{
    return accumulate(vect.begin(), vect.end(), Type());
}
```

Overloading templates does not have to restrict itself to the function's parameter list. The template's type parameter list itself may also be overloaded. The last definition of the `add()` template allows us to specify a `std::vector` as its first argument, but no `deque` or `map`. Overloaded versions for those types of containers could of course be constructed, but where's the end to that? Instead, let's look for common characteristics of these containers, and if found, define an overloaded template function on these common characteristics. One common characteristic of the mentioned containers is that they all support `begin()` and `end()` members, returning iterators. Using this, we could define a template type parameter representing containers that must support these members. But mentioning a plain 'container type' doesn't tell us for what data type it has been instantiated. So we need a second template type parameter representing the container's data type, thus overloading the template's type parameter list. Here is the resulting overloaded version of the `add()` template:

```
template <typename Container, typename Type>
Type add(Container const &cont, Type const &init)
{
    return std::accumulate(cont.begin(), cont.end(), init);
}
```

With all these overloaded versions in place, we may now start the compiler to compile the following function:

```
using namespace std;

int main()
{
    vector<int> v;

    add(3, 4);          // 1 (see text)
    add(v);             // 2
    add(v, 0);          // 3
}
```

- With the first statement, the compiler recognizes two identical types, both `int`. It will therefore instantiate `add<int>()`, our very first definition of the `add()` template.
- With statement two, a single argument is used. Consequently, the compiler will look for an overloaded version of `add()` requiring but one argument. It finds the version expecting a `std::vector`, deducing that the template's type parameter must be `int`. It instantiates

  ```
  add<int>(std::vector<int> const &)
  ```

- With statement three, the compiler again encounters an argument list holding two arguments. However, the types of the arguments are different, so it cannot use the `add()` template's first definition. But it *can* use the last definition, expecting entities having different types. As a `std::vector` supports `begin()` and `end()`, the compiler is now able to instantiate the template function

```
add<std::vector<int>, int>(std::vector<int> const &, int const &)
```

Having defined `add()` using two different template type parameters, and a template function having a parameter list containing two parameters of these types, we've exhausted the possibilities to define an `add()` function template having a function parameter list showing two different types. Even though the parameter types are different, we're still able to define a template function `add()` as a template function merely returning the sum of two differently typed entities:

```
template <typename T1, typename T2>
T1 add(T1 const &lvalue, T2 const &rvalue)
{
    return lvalue + rvalue;
}
```

However, now we won't be able to instantiate `add()` using two differently typed arguments anymore: the compiler won't be able resolve the ambiguity. It cannot choose which of the two overloaded versions defining two differently typed function parameters to use:

```
int main()
{
    add(3, 4.5);
}
/*
    Compiler reports:

    error: call of overloaded ‘add(int, double)’ is ambiguous
    error: candidates are: Type add(const Container&, const Type&)
                                [with Container = int, Type = double]
    error:                 T1 add(const T1&, const T2&)
                                [with T1 = int, T2 = double]
*/
```

Consider once again the overloaded function accepting three arguments:

```
template <typename Type>
Type add(Type const &lvalue, Type const &mvalue, Type const &rvalue)
{
    return lvalue + mvalue + rvalue;
}
```

It may be considered as a disadvantage that only equally typed arguments are accepted by this function: e.g., three `ints`, three `doubles` or three `strings`. To remedy this, we define yet another overloaded version of the function, this time accepting arguments of any type. Of course, when calling this function we must make sure that `operator+()` is defined between them, but apart from that there appears to be no problem. Here is the overloaded version accepting arguments of any type:

```
template <typename Type1, typename Type2, typename Type3>
```

```
Type1 add(Type1 const &lvalue, Type2 const &mvalue, Type3 const &rvalue)
{
    return lvalue + mvalue + rvalue;
}
```

Now that we've defined these two overloaded versions, let's call `add()` as follows:

```
    add(1, 2, 3);
```

In this case, one might expect the compiler to report an ambiguity. After all, the compiler might select the former function, deducing that `Type == int`, but it might also select the latter function, deducing that `Type1 == int`, `Type2 == int` and `Type3 == int`. However, the compiler reports no ambiguity. The reason for this is the following: if an overloaded template function is defined using *more specialized* template type parameters (e.g., all equal types) than another (overloaded) function, for which more general template type parameters (e.g., all different) have been used, then the compiler will select the more specialized function over the more general function wherever possible.

As a rule of thumb: when overloaded versions of a template function are defined, each overloaded version must use a unique combination of template type parameters to avoid ambiguities when the templates are instantiated. Note that the *ordering* of template type parameters in the function's parameter list is not important. When trying to instantiate the following `binarg()` template, an ambiguity will occur:

```
template <typename T1, typename T2>
void binarg(T1 const &first, T2 const &second)
{}
// and:
template <typename T1, typename T2>
void binarg(T2 const &first, T1 const &second)  // exchange T1 and T2
{}
```

The ambiguity should come as no surprise. After all, template type parameters are just formal names. Their names (`T1`, `T2` or `Whatever`) have no concrete meanings whatsoever.

Finally, overloaded functions may be declared, either using plain declarations or instantiation declarations, and explicit template parameter types may also be used. For example:

- Declaring a template function `add()` accepting containers of a certain type:

  ```
  template <typename Container, typename Type>
  Type add(Container const &container, Type const &init);
  ```

- The same function, but now using an instantiation declaration (note that this requires that the compiler has already seen the template's definition):

  ```
  template int add<std::vector<int>, int>
                  (std::vector<int> const &vect, int const &init);
  ```

- To disambiguate among multiple possibilities detected by the compiler, explicit arguments may be used. For example:

  ```
  std::vector<int> vi;
  int sum = add<std::vector<int>, int>(vi, 0);
  ```

18.7 Specializing templates for deviating types

The initial `add()` template, defining two identically typed parameters works fine for all types sensibly supporting `operator+()` and a copy constructor. However, these assumptions are not always met. For example, when `char *`s are used, neither the `operator+()` nor the copy constructor is (sensibly) available. The compiler does not know this, and will try to instantiate the simple template function

```
template <typename Type>
Type add(Type const &t1, Type const &t2);
```

But it can't do so, since `operator+()` is not defined for pointers. In situations like these it is clear that a match between the template's type parameter(s) and the actually used type(s) is possible, but the standard implementation is senseless or produces errors.

To solve this problem a *template explicit specialization* may be defined. A template explicit specialization defines the template function for which a generic definition already exists, using specific actual template type parameters.

In the abovementioned case an explicit specialization is required for a `char const *`, but probably also for a `char *` type. Probably, as the compiler still uses the standard type-deducing process mentioned earlier. So, when our `add()` template function is specialized for `char *` arguments, then its return type *must* also be a `char *`, whereas it *must* be a `char const *` if the arguments are `char const *` values. In these cases the template type parameter `Type` will be deduced properly. With `Type == char *`, for example, the head of the instantiated function becomes:

```
    char *add(char *const &t1, char *const &t2)
```

If this is considered undesirable, an *overloaded* version could be designed expecting pointers. The following template function definition expects two (`const`) pointers, and returns a non-const pointer:

```
template <typename T>
T *add(T const *t1, T const *t2)
{
    std::cout << "Pointers\n";
    return new T;
}
```

But we might still not be where we want to be, as *this* overloaded version will now only accept pointers to constant `T` elements. Pointers to non-const `T` elements will not be accepted. At first sight it may come as a surprise that the compiler will not apply a qualification transformation. But there's no need for the compiler to do so: when non-const pointers are used the compiler will simply use the initial definition of the `add()` template function expecting any two arguments of equal types.

So do we have to define yet another overloaded version, expecting non-const pointers? It is possible, but at some point it should become clear that we're overshooting our goal. Like concrete functions and classes, templates should have well-described purposes. Trying to add overloaded template definitions to overloaded template definitions quickly turns the template into a kludge. Don't follow this approach. A better approach is probably to construct the template so that it fits its original purpose, make allowances for the occasional specific case, and to describe its purpose clearly in the template's documentation.

Nevertheless, there may be situations where a template explicit specialization may be worth considering. Two specializations for `const` and non-`const` pointers to characters might be considered for our `add()` template function. Template explicit specializations are constructed as follows:

- They start with the keyword `template`.
- Next, an empty set of angle brackets is written. This indicates to the compiler that there must be an *existing* template whose prototype matches the one we're about to define. If we err and there is no such template then the compiler reports an error like:

```
error: template-id 'add<char*>' for 'char* add(char* const&, char*
       const&)' does not match any template declaration
```

- Next the head of the function is defined, which must follow the same syntax as a template explicit instantiation declaration (see section 18.3.1): it must specify the correct returntype, function name, template type parameter explicitations, as well as the function's parameter list.
- The body of the function, definining the special implementation that is required for the special actual template parameter types.

Here are two explicit specializations for the template function `add()`, expecting `char *` and `char const *` arguments (note that the `const` still appearing in the first template specialization is unrelated to the specialized type (`char *`), but refers to the `const &` mentioned in the original template's definition. So, in this case it's a reference to a constant pointer to a `char`, implying that the `char`s may be modified):

```
template <> char *add<char *>(char * const &p1,
                                    char * const &p2)
{
    std::string str(p1);
    str += p2;
    return strcpy(new char[str.length() + 1], str.c_str());
}

template <> char const *add<char const *>(char const *const &p1,
                                    char const *const &p2)
{
    static std::string str;
    str = p1;
    str += p2;
    return str.c_str();
}
```

Template explicit specializations are normally included in the file containing the other template function's implementations.

A template explicit specialization can be declared in the usual way. I.e., by replacing its body with a semicolon.

Note in particular how important the pair of angle brackets are that follow the `template` keyword when declaring a template explicit specialization. If the angle brackets were omitted, we would have constructed a template instantiation declaration. The compiler would silently process it, at the expense of a somewhat longer compilation time.

When declaring a template explicit specialization (or when using an instantiation declaration) the explicit specification of the template type parameters can be omitted if the compiler is able to deduce these types from the function's arguments. As this is the case with the `char (const) *` specializations, they could also be declared as follows:

```
template <> char const *add(char const *const &p1,
                                    char const *const &p2);
template <> char const *add(char const *const &p1,
                                    char const *const &p2);
```

In addition, `template <>` could be omitted. However, this would remove the template character from the declaration, as the resulting declaration is now nothing but a plain function declaration. This is not an error: template functions and non-template functions may overload each other. Ordinary functions are not as restrictive as template functions with respect to allowed type conversions. This could be a reason to overload a template with an ordinary function every once in a while.

18.8 The template function selection mechanism

When the compiler encounters a function call, it must decide which function to call when overloaded functions are available. In this section this function selection mechanism is described.

In our discussion, we assume that we ask the compiler to compile the following `main()` function:

```
int main()
{
    double x = 12.5;
    add(x, 12.5);
}
```

Furthermore we assume that the compiler has seen the following six function declarations when it's about to compile `main()`:

```
template <typename Type>                                    // function 1
Type add(Type const &lvalue, Type const &rvalue);

template <typename Type1, typename Type2>                   // function 2
Type1 add(Type1 const &lvalue, Type2 const &rvalue);

template <typename Type1, typename Type2, typename Type3>   // function 3
Type1 add(Type1 const &lvalue, Type1 const &mvalue, Type2 const &rvalue);

double add(float lvalue, double rvalue);                    // function 4
double add(std::vector<double> const &vd);                  // function 5
double divide(double lvalue, double rvalue);                // function 6
```

The compiler, having read `main()`'s statement, must now decide which function must actually be called. It proceeds as follows:

- First, a set of *candidate functions* is constructed. This set contains all functions that:
  - are visible at the point of the call;
  - have the same names as the called function.

As function 6 has a different name, it is removed from the set. The compiler is left with a set of five candidate functions: 1 until 5.

- 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 implies that the number of arguments must match the number of parameters of the viable functions.
- As functions 3 and 5 have different numbers of parameters they are removed from the set.
- Now let's 'play compiler' to decide among the remaining functions 1, 2 and 4. This is done by assigning *penalty points* to the remaining functions. Eventually the function having the smallest score will be selected. A point is assigned for every standard argument deduction process transformation that is required (so, for every *lvalue-, qualification-*, or *derived-to-base class* transformation that is applied).
- Eventually multiple functions might emerge at the top. Even though we have a draw in this case, the compiler will not always report an ambiguity. As we've seen before, a more specialized function is selected over a more general function. So, if a template explicit specialization and its more general variant appear at the top, the specialization is selected. Similarly, a concrete function will be selected over a template function (but remember: only if both appear at the top of the ranking process).
- As a rule of thumb we have:
  - when there are multiple viable functions at the top of the set of viable functions, then the plain function template instantiations are removed;
  - if multiple functions remain, template explicit specializations are removed;
  - if only one function remains, it is selected;
  - otherwise, the compiler can't decide and reports an error: the call is ambiguous.

Now we'll apply the above procedure to the viable functions 1, 2 and 4. As we will find function 1 to contain a slight complication, we'll start with function 2.

- Function 2 has prototype:

  ```
  template <typename T1, typename T2>
  T1 add(T1 const &a, T2 const &b);
  ```

  The function is called as `add(x, 12.5)`. As `x` is a `double` both `T &x` and `T const &x` would be acceptable, albeit that `T const &x` will require a qualification transformation. Since the function's prototype uses `T const &` a qualification transformation is needed. The function is charged 1 point, and tf(T1) is now determined as `double`.

  Next, 12.5 is recognized as a `double` as well (note that `float` constants are recognized by their 'F' suffix, e.g., 12.5F), and it is also a constant value. So, without transformations, we find `12.5 == T2 const &` and at no charge `T1` is recognized as `double` as well.

- Function 4 has prototype:

  ```
  double add(float lvalue, double rvalue);
  ```

  Although it is called as `add(x, 12.5)` with `x` being of type `double`; but a standard conversion exists from type `double` to type `float`. Furthermore, 12.5 is a `double`, which can be used to initialize `rvalue`.

Thus, at this point we could ask the compiler to select among:

```
add(double const &, double const &b);
```

and

```
add(float, double);
```

This does not involve 'template function selection' since the first one has already been determined. As the first function doesn't require any standard conversion at all, it is selected, since a perfect match is selected over one requiring a standard conversion.

As an intermezzo you are invited to take a closer look at this process by defining `float x` instead of `double x`, or by defining `add(float x, double x)` as add(double x, double x): in these cases the template function has the same prototype as the non-template function, and so the non-template function is selected since it's a more specific function. Earlier we've seen that process in action when redefining `ostream::operator»(ostream &os, string &str)` as a non-template function.

Now it's time to go back to template function 1.

- Function 1 has prototype:

  ```
  template <typename T>
  T add(T const &t1, T const &t2);
  ```

  Once again we call `add(x, 12.5)` and will deduce template types. In this case there's only one template type parameter T. Let's start with the first parameter:

  – The argument `x` is of type `double`, so both `T &x` and `T const &x` are acceptable. Acoording to the function's parameter list `T const &x` must be used, which requires a qualification transformation. So we'll charge the function 1 point and T is determined as `double`. This results in the instantiation of

    ```
    add(double const &t1, double const &t2)
    ```

    allowing us to call, at the expense of 1 point, `add(x, 12.5)`.

  But we can do better by starting our deduction process at the *second* parameter:

  – Since 12.5 is a constant `double` value we see that `12.5 == T const &`. So we conclude (free of charge) that T is `double`. Our function becomes

    ```
    add(double const &t1, double const &t2)
    ```

    allowing us to call `add(x, 12.5)`.

Earlier this section, we preferred function 2 over function 4. Function 2 is a template function that required one qualification transformation. Function 1, on the other hand, did not require any transformation at all, so it emerges as the function to be used.

As an exercise, feed the above six declarations and `main()` to the compiler and wait for the linker errors: the linker will complain that the (template) function

```
double add<double>(double const&, double const&)
```

is an undefined reference.

18.9 Compiling template definitions and instantiations

Consider the following definition of the `add()` template function:

```
template <typename Container, typename Type>
Type add(Container const &container, Type init)
{
    return std::accumulate(container.begin(), container.end(), init);
}
```

In this template definition, `std::accumulate()` is called, using `container`'s `begin()` and `end()` members.

The calls `container.begin()` and `container.end()` are said to *depend on template type parameters*. The compiler, not having seen `container`'s interface, cannot check whether `container` will actually have members `begin()` and `end()` returning input iterators, as required by `std::accumulate`.

On the other hand, `std::accumulate()` itself is a function call which is independent of any template type parameter. Its *arguments* are dependent of template parameters, but the function call itself isn't. Statements in a template's body that are independent of template type parameters are said *not to depend on template type parameters*.

When the compiler reads a template definition, it will verify the syntactical correctness of all statements not depending on template type parameters. I.e., it must have seen all class definitions, all type definitions, all function declarations etc., that are used in the statements not depending on the template's type parameters. If this condition isn't met, the compiler will not accept the template's definition. Consequently, when defining the above template, the header file `numeric` must have been included first, as this header file declares `std::accumulate()`.

On the other hand, with statements depending on template type parameters the compiler cannot perform these extensive checks, as it has, for example, no way to verify the existence of a member `begin()` for the as yet unspecified type `Container`. In these cases the compiler will perform superficial checks, assuming that the required members, operators and types will eventually become available.

The location in the program's source where the template is instantiated is called its *point of instantiation*. At the point of instantiation the compiler will deduce the actual types of the template's type parameters. At that point it will check the syntactical correctness of the template's statements that depend on template type parameters. This implies that *only at the point of instantiation* the required declarations must have been read by the compiler. As a rule of thumb, make sure that all required declarations (usually: header files) have been read by the compiler at every point of instantiation of the template. For the template's definition itself a more relaxed requirement can be formulated. When the definition is read only the declarations required for statements *not* depending on the template's type parameters must be known.

18.10 Summary of the template declaration syntax

In this section the basic syntactical constructions when declaring templates are summarized. When *defining* templates, the terminating semicolon should be replaced by a function body. However, not every template declaration may be converted into a template definition. If a definition may be provided it is explicitly mentioned.

- A plain template declaration (a definition is possible):

```
template <typename Type1, typename Type2>
void function(Type1 const &t1, Type2 const &t2);
```

- A template instantiation declaration (no definition):

```
template
void function<int, double>(int const &t1, double const &t2);
```

- A template using explicit types (no definition):

```
void (*fp)(double, double) = function<double, double>;
void (*fp)(int, int) = function<int, int>;
```

- A template specialization (a definition is possible):

```
template <>
void function<char *, char *>(char *const &t1, char *const &t2);
```

- A template declaration declaring friend template functions within template classes (covered in section 19.8):

```
friend void function<Type1, Type2>(parameters);
```

Chapter 19

Template classes

Like function templates, templates can be constructed for complete classes. A template class can be considered when the class should be able to handle different types of data. Template classes are frequently used in **C++**: chapter 12 covered general data structures like `vector, stack` and `queue`, defined 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, operating on a particular data type, only the data type needs to be specified when the template class object is defined or declared, e.g., `stack<int> iStack`.

Below the construction of template classes is discussed. In a sense, template classes compete with object oriented programming (cf. chapter 14), where a mechanism somewhat similar to templates is seen. Polymorphism allows the programmer to postpone the definitions of algorithms, by deriving classes from a base class in which the algorithm is only partially implemented, while the data upon which the algorithms operate may first be defined in derived classes, together with member functions that were defined as pure virtual functions in the base class to handle the data. On the other hand, templates allow the programmer to postpone the specification of the data upon which the algorithms operate. This is most clearly seen with the abstract containers, completely specifying the algorithms but at the same time leaving the data type on which the algorithms operate completely unspecified.

The correspondence between template classes and polymorphic classes is well-known. In their book **C++ Coding Standards** (Addison-Wesley, 2005) Sutter and Alexandrescu (2005) refer to *static polymorphism* and *dynamic polymorphism*. *Dynamic* polymorphism is what we use when overriding virtual members: Using the *vtable* construction the function that's actually called depends on the type of object a (base) class pointer points to. *Static* polymorphism is used when templates are used: depending on the actual types, the compiler *creates* the code, compile time, that's appropriate for those particular types. There's no need to consider static and dynamic polymorphism as mutually exlusive variants of polymorphism. Rather, both can be used together, combining their strengths. A warning is in place, though. When a template class defines virtual members *all* virtual members are instantiated for every instantiated type. This has to happen, since the compiler must be able to construct the class's `vtable`.

Generally, template classes are easier to use. It is certainly easier to write `stack<int> istack` to create a stack of `ints` than to derive a new class `Istack:  public stack` and to implement all necessary member functions 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 (but see also section 19.9).

19.1 Defining template classes

Now that we've covered the construction of template functions, we're ready for the next step: constructing template classes. Many useful template classes already exist. Instead of illustrating how an existing template class was constructed, let's discuss the construction of a useful new template class.

In chapter 17 we've encountered the `auto_ptr` class (section 17.3). The `auto_ptr`, also called *smart pointer*, allows us to define an object, acting like a pointer. Using `auto_ptrs` rather than plain pointers we not only ensure proper memory management, but we may also prevent memory leaks when objects of classes using pointer data-members cannot completely be constructed.

The one disadvantage of `auto_ptrs` is that they can only be used for single objects and not for pointers to arrays of objects. Here we'll construct the template class `FBB::auto_ptr`, behaving like `auto_ptr`, but managing a pointer to an array of objects.

Using an existing class as our point of departure also shows an important design principle: it's often easier to construct a template (function or class) from an existing template than to construct the template completely from scratch. In this case the existing `std::auto_ptr` acts as our model. Therefore, we want to provide the class with the following members:

- Constructors to create an object of the class `FBB::auto_ptr`;
- A destructor;
- An overloaded `operator=()`;
- An `operator[]()` to retrieve and reassign the elements given their indices.
- All other members of `std::auto_ptr`, with the exception of the dereference operator (`operator*()`), since our `FBB::auto_ptr` object will hold multiple objects, and although it would be entirely possible to define it as a member returning a reference to the first element of its array of objects, the member `operator+(int index)`, returning the address of object `index` would most likely be expected too. These extensions of `FBB::auto_ptr` are left as exercises to the reader.

Now that we have decided which members we need, the class interface can be constructed. Like template functions, a template class definition begins with the keyword `template`, which is also followed by a non-empty list of template type and/or non-type parameters, surrounded by angle brackets. The `template` keyword followed by the template parameter list enclosed in angle brackets is called a *template announcement* in the **C++** Annotations. In some cases the template announcement's parameter list may be empty, leaving only the angle brackets.

Following the template announcement the class interface is provided, in which the formal template type parameter names may be used to represent types and constants. The class interface is constructed as usual. It starts with the keyword `class` and ends with a semicolon.

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 `Type const &`, rather than `Type`, to prevent unnecessary copying of large data structures. Template class constructors should use member initializers rather than member assignment within the body of the constructors, again to prevent double assignment of composed objects: once by the default constructor of the object, once by the assignment itself.

Here is our initial version of the class `FBB::auto_ptr` showing all its members:

```
namespace FBB
```

```
{
    template <typename Data>
    class auto_ptr
    {
        Data *d_data;

        public:
            auto_ptr();
            auto_ptr(auto_ptr<Data> &other);
            auto_ptr(Data *data);
            ~auto_ptr();
            auto_ptr<Data> &operator=(auto_ptr<Data> &rvalue);
            Data &operator[](size_t index);
            Data const &operator[](size_t index) const;
            Data *get();
            Data const *get() const;
            Data *release();
            void reset(Data *p = 0);
        private:
            void destroy();
            void copy(auto_ptr<Data> &other);
            Data &element(size_t idx) const;
    };

    template <typename Data>
    inline auto_ptr<Data>::auto_ptr()
    :
        d_data(0)
    {}

    template <typename Data>
    inline auto_ptr<Data>::auto_ptr(auto_ptr<Data> &other)
    {
        copy(other);
    }

    template <typename Data>
    inline auto_ptr<Data>::auto_ptr(Data *data)
    :
        d_data(data)
    {}

    template <typename Data>
    inline auto_ptr<Data>::~auto_ptr()
    {
        destroy();
    }

    template <typename Data>
    inline Data &auto_ptr<Data>::operator[](size_t index)
    {
        return d_data[index];
    }
```

```
template <typename Data>
inline Data const &auto_ptr<Data>::operator[](size_t index) const
{
    return d_data[index];
}

template <typename Data>
inline Data *auto_ptr<Data>::get()
{
    return d_data;
}

template <typename Data>
inline Data const *auto_ptr<Data>::get() const
{
    return d_data;
}

template <typename Data>
inline void auto_ptr<Data>::destroy()
{
    delete[] d_data;
}

template <typename Data>
inline void auto_ptr<Data>::copy(auto_ptr<Data> &other)
{
    d_data = other.release();
}

template <typename Data>
auto_ptr<Data> &auto_ptr<Data>::operator=(auto_ptr<Data> &rvalue)
{
    if (this != &rvalue)
    {
        destroy();
        copy(rvalue);
    }
    return *this;
}

template <typename Data>
Data *auto_ptr<Data>::release()
{
    Data *ret = d_data;
    d_data = 0;
    return ret;
}

template <typename Data>
void auto_ptr<Data>::reset(Data *ptr)
{
    destroy();
    d_data = ptr;
```

```
    }

} // FBB
```

The class interface shows the following features:

- If it is assumed that the template type `Data` is an ordinary type, the class interface appears to have no special characteristics at all. It looks like any old class interface. This is generally true. Often a template class can easily be constructed after having constructed the class for one or two concrete types, followed by an abstraction phase changing all necessary references to concrete data types into generic data types, which then become the template's type parameters.
- At closer inspection, some special characteristics can actually be discerned. The parameters of the class's copy constructor and overloaded assignment operators aren't references to plain `auto_ptr` objects, but rather references to `auto_ptr<Data>` objects. Template class objects (or their references or pointers) *always* require the template type parameters to be specified.
- Different from the standard design of copy constructors and overloaded assignment operators, their parameters are *non*-`const` references. This has nothing to do with the class being a template class, but is a consequence of `auto_ptr`'s design itself: both the copy constructor and the overloaded assignment operator take the other's object's pointer, effectively changing the other object into a 0-pointer.
- Like ordinary classes, members can be defined *inline*. Actually, *all* template class members are defined inline (when using precompiled templates *precompiled templates* this doesn't change; it only means that the compiler has reorganized the template definition so that it can process the definition faster). As noted in section 6.3, the definition may be put inside the class interface or outside (i.e., following) the class interface. As a rule of thumb the same design principles should be followed here as with concrete classes: they should be defined below the interface to keep the interface clean and readable. Long implementations in the interface tend to obscure the interface itself.
- When objects of a template class are instantiated, the definitions of all the template's member functions that are used (but *only* those) must have been seen by the compiler. Although that characteristic of templates could be refined to the point where each definition is stored in a separate template function definition file, including only the definitions of the template functions that are actually needed, it is hardly ever done that way (even though it would speed up the required compilation time). Instead, the usual way to define template classes is to define the interface, defining some functions inline, and to define the remaining template functions immediately below the template class's interface.
- Beside the dereference operator (`operator*()`), the well-known pair of `operator[]()` members are defined. Since the class receives no information about the size of the array of objects, these members cannot support array-bound checking.

Let's have a look at some of the member functions defined beyond the class interface. Note in particular:

- The definition below the interface is the actual template definition. Since it is a definition it must start with a `template phrase`. The function's *declaration* must also start with a `template` phrase, but that is implied by the interface itself, which already provides the required phrase at its very beginning;
- Wherever `auto_ptr` is mentioned in the implementation, the template's type parameter is mentioned as well. This is obligatory.

Some remarks about specific members:

- The advised copy() and destroy() members (see section 7.5.1) are very simple, but were added to the implementation to promote standardization of classes containing pointer members.
- The overloaded assignment constructor still has to check for auto-assignment.

Now that the class has been defined, it can be used. To use the class, its object must be instantiated for a particular data type. The example defines a new std::string array, storing all command-line arguments. Then, the first command-line argument is printed. Next, the auto_ptr object is used to initialize another auto_ptr of the same type. It is shown that the original auto_ptr now holds a 0-pointer, and that the second auto_ptr object now holds the command-line arguments:

```
#include <iostream>
#include <algorithm>
#include <string>
#include "autoptr.h"
using namespace std;

int main(int argc, char **argv)
{
    FBB::auto_ptr<string> sp(new string[argc]);
    copy(argv, argv + argc, sp.get());

    cout << "First auto_ptr, program name: " << sp[0] << endl;

    FBB::auto_ptr<string> second(sp);

    cout << "First auto_ptr, pointer now: " << sp.get() << endl;
    cout  << "Second auto_ptr, program name: " << second[0] << endl;

    return 0;
}
/*
    Generated output:

    First auto_ptr, program name: a.out
    First auto_ptr, pointer now: 0
    Second auto_ptr, program name: a.out
*/
```

19.1.1 Default template class parameters

Different from template functions, template parameters of template classes may be given default values. This holds true both for template type- and template non-type parameters. If a template class is instantiated without specifying arguments for its template parameters, and if default template parameter values were defined, then the defaults are used. When defining such defaults keep in mind that the defaults should be suitable for the majority of instantiations of the class. E.g., for the template class FBB::auto_ptr the template's type parameter list could have been altered by specifying int as its default type:

```
        template <typename Data = int>
```

Even though default arguments can be specified, the compiler must still be informed that object definitions refer to templates. So, when instantiating template class objects for which default parameter values have been defined the type specifications may be omitted, but the angle brackets must remain. So, assuming a default type for the `FBB::auto_ptr` class, an object of that class may be defined as:

```
    FBB::auto_ptr<> intAutoPtr;
```

No defaults must be specified for template members defined outside of their class interface. Template functions, even template member functions, cannot specify default parameter values. So, the definition of, e.g., the `release()` member will always begin with the same `template` specification:

```
    template <typename Data>
```

When a template class uses multiple template parameters, all may be given default values. However, like default function arguments, once a default value is used, all remaining parameters must also use their default values. A template type specification list may not start with a comma, nor may it contain multiple consecutive commas.

19.1.2 Declaring template classes

Template classes may also be *declared*. This may be useful in situations where forward class declarations are required. To declare a template class, replace its interface (the part between the curly braces) by a semicolon:

```
namespace FBB
{
    template <typename Type>
    class auto_ptr;
}
```

Here default types may also be specified. However, default type values cannot be specified in both the declaration and the definition of a template class. As a rule of thumb default values should be omitted from *declarations*, as template class declarations are never used when instantiating objects, but only for the occasional forward reference. Note that this differs from default parameter value specifications for member functions in concrete classes. Such defaults should be specified in the member functions' declarations and *not* in their definitions.

19.1.3 Distinguishing members and types of formal class-types

Since a template type name may refer to any type, a template's type name might also refer to a template or a class itself. Let's assume a template class `Handler` defines a `typename Container` as its type parameter, and a data member storing the container's `begin()` iterator. Furthermore, the template class `Handler` has a constructor accepting any container supporting a `begin()` member. The skeleton of our class `Handler` could then be:

```
template <typename Container>
class Handler
{
```

```
        Container::const_iterator d_it;

        public:
            Handler(Container const &container)
            :
                d_it(container.begin())
            {}
    };
```

What were the considerations we had in mind when designing this class?

- The typename `Container` represents any container supporting iterators.
- The container presumably supports a member `begin()`. The initialization `d_it(container.begin())` clearly depends on the template's type parameter, so it's only checked for basic syntactical correctness.
- Likewise, the container presumably supports a type `const_iterator`, defined in the class `Container`. Since `container` is a `const` reference, the iterator returned by `begin()` is a `const_iterator` rather than a plain `iterator`.

Now, when instantiating a `Handler` using the following `main()` function we run into a compilation error:

```
    #include "handler.h"
    #include <vector>
    using namespace std;

    int main()
    {
        vector<int> vi;
        Handler<vector<int> > ph(vi);
    }
    /*
        Reported error:

    handler.h:4: error: syntax error before ‘;’ token
    */
```

Apparently the line

```
        Container::const_iterator d_it;
```

in the `Handler` class causes a problem. The problem is the following: when using template type parameters, a plain syntax check allows the compiler to decide that ‘`container`’ refers to a `Container` object. Such a `Container` might very well support a `begin()` member, hence `container.begin()` is syntactically correct. However, for a actual `Container` type that member `begin()` might not have been implemented. Of course, whether or not `begin()` has in fact been implemented will only be known by the time `Container`'s actual type has been specified.

On the other hand, note that the compiler is unable to determine what a `Container::const_iterator` is. The compiler takes the easy way out, and assumes `const_iterator` is a member of the as yet mysterious `Container`. Therefore, a plain syntax check clearly fails, as the statement

```
        Container::const_iterator d_it;
```

is always syntactically wrong when `const_iterator` is a member or enum-value of `Container`. Of course, *we* know better, since we have a type that is nested under the class `Container` in mind. The compiler, however, doesn't know that and before it has parsed the complete definition, it has already read `Container::const_iterator`. At that point the compiler has already made up its mind, assuming that `Container::const_iterator` will be a member, rather than a type.

That the compiler indeed assumes `X::a` is a member `a` of the class `X` is illustrated by the error message we get when we try to compile `main()` using the following implementation of `Handler`'s constructor:

```
Handler(Container const &container)
:
    d_it(container.begin())
{
    size_t x = Container::ios_end;
}
/*
    Reported error:

    error: 'ios_end' is not a member of type 'std::vector<int,
            std::allocator<int> >'
*/
```

In cases like these, where the intent is to refer to a *type* defined in (or depending on) a template class like `Container`, this must explicitly be indicated to the compiler, using the `typename` keyword. Here is the `Handler` class once again, now using `typename`:

```
template <typename Container>
class Handler
{
    typename Container::const_iterator d_it;
    public:
        Handler(Container const &container);
};

template <typename Container>
inline Handler<Container>::Handler(Container const &container)
:
    d_it(container.begin())
{}
```

Now `main()` will compile correctly. The `typename` keyword may also be required when specifying the proper return types of template class member functions returning values of nested types defined within the template class. Section 19.11.2 provides an example of this situation.

19.1.4 Non-type parameters

As we've seen with template functions, template parameters are either template type parameters or template non-type parameters. Template classes may also define non-type parameters. Like the non-const parameters used with template functions they must be constants whose values are known by the time an object is instantiated.

However, their values are not deduced by the compiler using arguments passed to constructors. Assume we modify the template class `FBB::auto_ptr` so that it has an additional non-type parameter `size_t Size`. Next we use this `Size` parameter in a new constructor defining an array of `Size` elements of type `Data` as its parameter. The new `FBB::auto_ptr` template class becomes (showing only the relevant constructors; note the two template type parameters that are now required, e.g., when specifying the type of the copy constructor's parameter):

```
namespace FBB
{
    template <typename Data, size_t Size>
    class auto_ptr
    {
        Data *d_data;
        size_t d_n;

        public:
            auto_ptr(auto_ptr<Data, Size> &other);
            auto_ptr(Data2 *data);
            auto_ptr(Data const (&arr)[Size]);
            ...
    };

    template <typename Data, size_t Size>
    inline auto_ptr<Data, Size>::auto_ptr(Data const (&arr)[Size])
    :
        d_data(new Data2[Size]),
        d_n(Size)
    {
        std::copy(arr, arr + Size, d_data);
    }
}
```

Unfortunately, this new setup doesn't satisfy our needs, as the values of template non-type parameters are not deduced by the compiler. When the compiler is asked to compile the following `main()` function it reports a mismatch between the required and actual number of template parameters:

```
int main()
{
    int arr[30];

    FBB::auto_ptr<int> ap(arr);
}
/*
    Error reported by the compiler:

    In function ‘int main()’:
        error: wrong number of template arguments (1, should be 2)
        error: provided for ‘template<class Data, size_t Size>
                class FBB::auto_ptr’
*/
```

Making `Size` into a non-type parameter having a default value doesn't work either. The compiler will use the default, unless explicitly specified otherwise. So, reasoning that `Size` can be 0 unless we need another value, we might specify `size_t Size = 0` in the templates parameter type list. However, this causes a mismatch between the default value 0 and the actual size of the array `arr` as defined in the above `main()` function. The compiler, using the default value, reports:

```
In instantiation of ‘FBB::auto_ptr<int, 0>’:
...
error: creating array with size zero (‘0’)
```

So, although template classes may use non-type parameters, they must be specified like the type parameters when an object of the class is defined. Default values can be specified for those non-type parameters, but then the default will be used when the non-type parameter is left unspecified.

Note that *default template parameter values* (either type or non-type template parameters) may *not* be used when template member functions are defined outside the class interface. Template function definitions (and thus: template class member functions) may not be given default template (non) type parameter values. If default template parameter values are to be used for template class members, they have to be specified in the class interface.

Similar to non-type parameters of template functions, non-type parameters of template classes may only be specified as constants:

- Global variables have constant addresses, which can be used as arguments for non-type parameters.
- Local and dynamically allocated variables have addresses that are not known by the compiler when the source file is compiled. These addresses can therefore not be used as arguments for non-type parameters.
- Lvalue transformations are allowed: if a pointer is defined as a non-type parameter, an array name may be specified.
- Qualification conversions are allowed: a pointer to a non-const object may be used with a non-type parameter defined as a `const` pointer.
- Promotions are allowed: a constant of a ‘narrower’ data type may be used 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 `size_t` parameter is specified, an `int` may be used too.
- Variables cannot be used to specify template non-type parameters, as their values are not constant expressions. Variables defined using the `const` modifier, however, may be used, as their values never change.

Although our attempts to define a constructor of the class `FBB::auto_ptr` accepting an array as its argument, allowing us to use the array’s size within the constructor’s code has failed so far, we’re not yet out of options. In the next section an approach will be described allowing us to reach our goal, after all.

19.2 Member templates

Our previous attempt to define a template non-type parameter which is initialized by the compiler to the number of elements of an array failed because the template’s parameters are not implicitly deduced when a constructor is called, but they are explicitly specified, when an object of the template class is defined. As the parameters are specified just before the template's constructor is called, there's nothing to deduce anymore, and the compiler will simply use the explicitly specified template arguments.

On the other hand, when template *functions* are used, the actual template parameters are deduced from the arguments used when calling the function. This opens an approach route to the solution of our problem. If the constructor itself is made into a member which itself is a template function (containing a template announcement of its own), then the compiler will be able to deduce the non-type parameter's value, without us having to specify it explicitly as a template class non-type parameter.

Member functions (or classes) of template classes which themselves are templates are called *member templates*. Member templates are defined in the same way as any other template, including the `template <typename ...>` header.

When converting our earlier `FBB::auto_ptr(Data const (&array)[Size])` constructor into a member template we may use the template class's `Data` type parameter, but must provide the member template with a non-type parameter of its own. The class interface is given the following additional member declaration:

```
template <typename Data>
class auto_ptr
{
    ...
    public:
        template <size_t Size>
        auto_ptr(Data const (&arr)[Size]);
    ...
};
```

and the constructor's implementation becomes:

```
template <typename Data>
template <size_t Size>
inline auto_ptr<Data>::auto_ptr(Data const (&arr)[Size])
:
    d_data(new Data[Size]),
    d_n(Size)
{
    std::copy(arr, arr + Size, d_data);
}
```

Member templates have the following characteristics:

- Normal access rules apply: the constructor can be used by the general program to construct an `FBB::auto_ptr` object of a given data type. As usual for template classes, the data type must be specified when the object is constructed. To construct an `FBB::auto_ptr` object from the array `int array[30]` we define:

  ```
  FBB::auto_ptr<int> object(array);
  ```

- Any member can be defined as a member template, not just a constructor.
- When a template member is defined below its class, the template class parameter list must precede the template function parameter list of the template member. Furthermore:

- The member should be defined inside its proper namespace environment. The organization within files defining template classes within a namespace should therefore be:

```
namespace SomeName
{
    template <typename Type, ...>   // template class definition
    class ClassName
    {
        ...
    };

    template <typename Type, ...>   // non-inline member definition(s)
    ClassName<Type, ...>::member(...)
    {
        ...
    }
}                                   // namespace closed
```

- Two template announcements must be used: the template class's template announcement is specified first, followed by the member template's template announcement.
- The definition itself must specify the member template's proper scope: the member template is defined as a member of the class `FBB::auto_ptr`, instantiated for the formal template parameter type `Data`. Since we're already inside the namespace `FBB`, the function header starts with `auto_ptr<Data>::auto_ptr`.
- The formal template parameter names in the declaration and implementation must be identical.

One small problem remains. When we're constructing an `FBB::auto_ptr` object from a fixed-size array the above constructor is not used. Instead, the constructor `FBB::auto_ptr<Data>::auto_ptr(Data *data)` is activated. As the latter constructor is not a member template, it is considered a more specialized version of a constructor of the class `FBB::auto_ptr` than the former constructor. Since both constructors accept an array the compiler will call `auto_ptr(Data *)` rather than `auto_ptr(Data const (&array)[Size])`. This problem can be solved by simply changing the constructor `auto_ptr(Data *data)` into a member template as well, in which case its template type parameter should be changed into '`Data`'. The only remaining subtlety is that template parameters of member templates may not shadow the template parameters of their class. Renaming `Data` into `Data2` takes care of this subtlety. Here is the (inline) definition of the `auto_ptr(Data *)` constructor, followed by an example in which both constructors are actually used:

```
template <typename Data>
template <typename Data2>           // data: dynamically allocated
inline auto_ptr<Data>::auto_ptr(Data2 *data)
:
    d_data(data),
    d_n(0)
{}
```

Calling both constructors in `main()`:

```
int main()
{
    int array[30];

    FBB::auto_ptr<int> ap(array);
```

```
        FBB::auto_ptr<int> ap2(new int[30]);

        return 0;
    }
```

19.3 Static data members

When static members are defined in template classes, they are instantiated for every new instantiation. 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
    {
        static int s_objectCounter;
    };
```

There will be *one* `TheClass<Type>::objectCounter` for each different `Type` specification. The following instantiates just one single static variable, shared among the different objects:

```
    TheClass<int> theClassOne;
    TheClass<int> theClassTwo;
```

Mentioning static members in interfaces does not mean these members are actually defined: they are only *declared* by their classes and must be *defined* separately. With static members of template classes this is not different. The definitions of static members are usually provided immediately following (i.e., below) the template class interface. The static member `s_objectCounter` will thus be defined as follows, just below its class interface:

```
    template <typename Type>                    // definition, following
    int TheClass<Type>::s_objectCounter = 0;    // the interface
```

In the above case, `s_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` must be used to define the static variable, as shown in the following example:

```
    template <typename Type>
    class TheClass
    {
        static TheClass *s_objectPtr;
    };

    template <typename Type>
    TheClass<Type> *TheClass<Type>::s_objectPtr = 0;
```

As usual, the definition can be read from the variable name back to the beginning of the definition: `s_objectPtr` of the class `TheClass<Type>` is a pointer to an object of `TheClass<Type>`.

Finally, when a static variable of a template's type parameter is defined, it should of course not be given the initial value 0. The default constructor (e.g., `Type()` will usually be more appropriate):

```
template <typename Type>                    // s_type's definition
Type TheClass<Type>::s_type = Type();
```

19.4 Specializing template classes for deviating types

Our earlier class `FBB::auto_ptr` can be used for many different types. Their common characteristic is that they can simply be assigned to the class's `d_data` member, e.g., using `auto_ptr(Data *data)`. However, this is not always as simple as it looks. What if `Data`'s actual type is `char *`? Examples of a `char **`, `data`'s resulting type, are well-known: `main()`'s `argv` and `envp`, for example are `char **` parameters.

It this special case we might not be interested in the mere reassignment of the constructor's parameter to the class's `d_data` member, but we might be interested in copying the complete `char **` structure. To realize this, template class specializations may be used.

Template class specializations are used in cases where template member functions cannot (or should not) be used for a particular actual template parameter type. In those cases specialized template members can be constructed, fitting the special needs of the actual type.

Template class member specializations are specializations of existing class members. Since the class members already exist, the specializations will *not* be part of the class interface. Rather, they are defined below the interface as members, redefining the more generic members using explicit types. Furthermore, as they are specializations of existing class members, their function prototypes must exactly match the prototypes of the member functions for which they are specializations. For our `Data = char *` specialization the following definition could be designed:

```
template <>
auto_ptr<char *>::auto_ptr(char **argv)
:
    d_n(0)
{
    char **tmp = argv;
    while (*tmp++)
        d_n++;
    d_data = new char *[d_n];

    for (size_t idx = 0; idx < d_n; idx++)
    {
        std::string str(argv[idx]);
        d_data[idx] =
            strcpy(new char[str.length() + 1], str.c_str());
    }
}
```

Now, the above specialization will be used to construct the following `FBB::auto_ptr` object:

```
int main(int argc, char **argv)
{
    FBB::auto_ptr<char *> ap3(argv);
```

```
        return 0;
    }
```

Although defining a template member specialization may allow us to use the occasional exceptional type, it is also quite possible that a single template member specialization is not enough. Actually, this is the case when designing the `char *` specialization, since the template's `destroy()` implementation is not correct for the specialized type `Data = char *`. When multiple members must be specialized for a particular type, then a complete template class specialization might be considered.

A completely specialized class shows the following characteristics:

- The template class specialization follows the generic template class definition. After all, it's a specialization, so the compiler must have seen what is being specialized.
- All the class's template parameters are given specific type names or (for the non-type parameters) specific values. These specific values are explicitly stated in a template parameter specification list (surrounded by angle brackets) which is inserted immediately following the template's class name.
- All the specialized template members specify the specialized types and values where the generic template parameters are used in the generic template definition.
- Not all the template's members *have* to be defined, but, to ensure generality of the specialization, *should* be defined. If a member is left out of the specialization, it can't be used for the specialized type(s).
- Additional members may be defined in the specialization. However, those that are defined in the generic template too must have corresponding members (using the same prototypes, albeit using the generic template parameters) in the generic template class definition. The compiler will not complain when additional members are defined, and will allow you to use those members with objects of the specialized template class.
- Member functions of specialized template classes may be defined within their specializing class or they may be declared in the specializing class. When they are only declared, then their definitition should be given below the specialized template class's interface. Such an implementation may *not* begin with a `template <>` announcement, but should immediately start with the member function's header.

Below a full specialization of the template class `FBB::auto_ptr` for the actual type `Data = char *` is given, illustrating the above characteristics. The specialization should be appended to the file already containing the generic template class. To reduce the size of the example members that are only declared may be assumed to have identical implementations as used in the generic template.

```
#include <iostream>
#include <algorithm>
#include "autoptr.h"

namespace FBB
{
    template<>
    class auto_ptr<char *>
    {
        char **d_data;
        size_t d_n;
```

```
    public:
        auto_ptr<char *>();
        auto_ptr<char *>(auto_ptr<char *> &other);
        auto_ptr<char *>(char **argv);

        // template <size_t Size>             NI
        // auto_ptr(char *const (&arr)[Size])

        ~auto_ptr();
        auto_ptr<char *> &operator=(auto_ptr<char *> &rvalue);
        char *&operator[](size_t index);
        char *const &operator[](size_t index) const;
        char **get();
        char *const *get() const;
        char **release();
        void reset(char **argv);
        void additional() const;    // just an additional public
                                    // member
    private:
        void full_copy(char **argv);
        void copy(auto_ptr<char *> &other);
        void destroy();
};

inline auto_ptr<char *>::auto_ptr()
:
    d_data(0),
    d_n(0)
{}

inline auto_ptr<char *>::auto_ptr(auto_ptr<char *> &other)
{
    copy(other);
}

inline auto_ptr<char *>::auto_ptr(char **argv)
{
    full_copy(argv);
}

inline auto_ptr<char *>::~auto_ptr()
{
    destroy();
}

inline void auto_ptr<char *>::reset(char **argv)
{
    destroy();
    full_copy(argv);
}

inline void auto_ptr<char *>::additional() const
{}
```

```
        inline void auto_ptr<char *>::full_copy(char **argv)
        {
            d_n = 0;
            char **tmp = argv;
            while (*tmp++)
                d_n++;
            d_data = new char *[d_n];

            for (size_t idx = 0; idx < d_n; idx++)
            {
                std::string str(argv[idx]);
                d_data[idx] =
                    strcpy(new char[str.length() + 1], str.c_str());
            }
        }

        inline void auto_ptr<char *>::destroy()
        {
            while (d_n--)
                delete d_data[d_n];
            delete[] d_data;
        }
    }
```

19.5 Partial specializations

In the previous section we've seen that it is possible to design template class specializations. It was shown that both template class members and complete template classes could be specialized. Furthermore, the specializations we've seen were specializing template type parameters.

In this section we'll introduce a variant of these specializations, both in number and types of template parameters that are specialized. *Partial specializations* may be defined for template classes having multiple template parameters. With partial specializations a subset (any subset) of template type parameters are given specific values.

Having discussed specializations of template type parameters in the previous section, we'll discuss specializations of non-type parameters in the current section. Partial specializations of template non-type parameters will be illustrated using some simple concepts defined in matrix algebra, a branch of linear algebra.

A matrix is commonly thought of as consisting of a table of a certain number of rows and columns, filled with numbers. Immediately we recognize an opening for using templates: the numbers might be plain `double` values, but they could also very well be complex numbers, for which our *complex container* (cf. section 12.4) might prove useful. Consequently, our template class should be given a `DataType` template type parameter, for which a concrete class can be specified when a matrix is constructed. Some simple matrices, using `double` values, are:

```
    1   0   0           An identity matrix,
    0   1   0           a 3 x 3 matrix.
    0   0   1

    1.2  0    0    0    A rectangular matrix,
    0.5  3.5  18   23   a 2 x 4 matrix.
```

```
1   2   4   8       A matrix of one row,
                    a 1 x 4 matrix, also known as a
                    'row vector' of 4 elements.
                    (column vectors are analogously defined)
```

Since matrices consist of a specific number of rows and columns (the *dimensions* of the matrix), which normally do not change when using matrices, we might consider specifying their values as template non-type parameters. Since the `DataType = double` selection will be used in the majority of cases, `double` can be selected as the template's default type. Since it's having a sensible default, the `DataType` template type parameter is put last in the template type parameter list. So, our template class `Matrix` starts off as follows:

```
template <size_t Rows, size_t Columns, typename DataType = double>
class Matrix
...
```

Various operations are defined on matrices. They may, for example be added, subtracted or multiplied. We will not focus on these operations here. Rather, we'll concentrate on a simple operation: computing marginals and sums. The row marginals are obtained by computing, for each row, the sum of all its elements, putting these `Rows` sum values in corresponding elements of a column vector of `Rows` elements. Analogously, column marginals are obtained by computing, for each column, the sum of all its elements, putting these `Columns` sum values in corresponding elements of a row vector of `Columns` elements. Finally, the sum of the elements of a matrix can be computed. This sum is of course equal to the sum of the elements of its marginals. The following example shows a matrix, its marginals, and its sum:

```
            matrix:         row
                            marginals:

            1   2   3        6
            4   5   6       15

column      5   7   9       21  (sum)
marginals
```

So, what do we want our template class to offer?

- It needs a place to store its matrix elements. This can be defined as an array of '`Rows`' rows each containing '`Columns`' elements of type `DataType`. It can be an array, rather than a pointer, since the matrix' dimensions are known *a priori*. Since a vector of `Columns` elements (a *row* of the matrix), as well as a vector of `Row` elements (a *column* of the matrix) is often used, *typedefs* could be used by the class. The class interface's initial section therefore contains:

```
typedef Matrix<1, Columns, DataType>    MatrixRow;
typedef Matrix<Rows, 1, DataType>       MatrixColumn;

MatrixRow d_matrix[Rows];
```

- It should offer constructors: a default constructor and, for example, a constructor initializing the matrix from a stream. No copy constructor is required, since the default copy constructor performs its task properly. Analogously, no overloaded assignment operator or destructor is required. Here are the constructors, defined in the public section:

```
template <size_t Rows, size_t Columns, typename DataType>
```

```
Matrix<Rows, Columns, DataType>::Matrix()
{
    std::fill(d_matrix, d_matrix + Rows, MatrixRow());
}

template <size_t Rows, size_t Columns, typename DataType>
Matrix<Rows, Columns, DataType>::Matrix(std::istream &str)
{
    for (size_t row = 0; row < Rows; row++)
        for (size_t col = 0; col < Columns; col++)
            str >> d_matrix[row][col];
}
```

- The class's `operator[]()` member (and its `const` variant) only handles the first index, returning a reference to a complete `MatrixRow`. How to handle the retrieval of elements in a `MatrixRow` will be covered shortly. To keep the example simple, no array bound check has been implemented:

```
template <size_t Rows, size_t Columns, typename DataType>
Matrix<1, Columns, DataType>
&Matrix<Rows, Columns, DataType>::operator[](size_t idx)
{
    return d_matrix[idx];
}
```

- Now we get to the interesting parts: computing marginals and the sum of all elements in a `Matrix`. Considering that marginals are vectors, either a `MatrixRow`, containing the column marginals, a `MatrixColumn`, containing the row marginals, or a single value, either computed as the sum of a vector of marginals, or as the value of a `1 x 1` matrix, initialized from a generic `Matrix`, we can now construct *partial specializations* to handle `MatrixRow` and `MatrixColumn` objects, and a partial specialization handling `1 x 1` matrices. Since we're about to define these specializations, we can use them when computing marginals and the matrix' sum of all elements. Here are the implementations of these members:

```
template <size_t Rows, size_t Columns, typename DataType>
Matrix<1, Columns, DataType>
Matrix<Rows, Columns, DataType>::columnMarginals() const
{
    return MatrixRow(*this);
}

template <size_t Rows, size_t Columns, typename DataType>
Matrix<Rows, 1, DataType>
Matrix<Rows, Columns, DataType>::rowMarginals() const
{
    return MatrixColumn(*this);
}

template <size_t Rows, size_t Columns, typename DataType>
DataType Matrix<Rows, Columns, DataType>::sum()  const
{
    return rowMarginals().sum();
}
```

Template class *partial specializations* may be defined for any (subset) of template parameters. They can be defined for template type parameters and for template non-type parameters alike. Our first partial specialization defines the special case where we construct a row of a generic `Matrix`, specifically aiming at (but not restricted to) the construction of column marginals. Here is how such a partial specialization is constructed:

- The partial specialization starts by defining all template type parameters which are *not* specialized in the partial specialization. This partial specialization template announcement cannot specify any defaults (like `DataType = double`), since the defaults have already been specified by the generic template class definition. Furthermore, the specialization *must* follow the definition of the generic template class definition, or the compiler will complain that it doesn't know what class is being specialized. Following the template announcement, the class interface starts. Since it's a template class (partial) specialization, the class name is followed by a template type parameter list specifying concrete values or types for all template parameters specified in this specialization, and using the template's generic (non-)type names for the remaining template parameters. In the `MatrixRow` specialization `Rows` is specified as 1, since we're talking here about one single row. Both `Columns` and `DataType` remain to be specified. So, the `MatrixRow` partial specialization starts as follows:

```
template <size_t Columns, typename DataType>  // no default specified
class Matrix<1, Columns, DataType>
```

- A `MatrixRow` contains the data of a single row. So it needs a data member storing `Columns` values of type `DataType`. Since `Columns` is a constant value, the `d_row` data member can be defined as an array:

```
    DataType d_column[Columns];
```

- The constructors require some attention. The default constructor is simple. It merely initializes the `MatrixRow`'s data elements, using `DataType`'s default constructor:

```
template <size_t Columns, typename DataType>
Matrix<1, Columns, DataType>::Matrix()
{
    std::fill(d_column, d_column + Columns, DataType());
}
```

  However, we also need a constructor initializing a `MatrixRow` object with the column marginals of a generic `Matrix` object. This requires us to provide the constructor with a non-specialized `Matrix` parameter. In cases like this, the rule of thumb is to define a member template allowing us to keep the general nature of the parameter. Since the generic `Matrix` template requires three template parameters, two of which are already provided by the template specialization, the third parameter must be specified in the member template's template announcement. Since this parameter refers to the generic matrix' number of rows, let's simply call it `Rows`. Here then, is the definition of the second constructor, initializing the `MatrixRow`'s data with the column marginals of a generic `Matrix` object:

```
template <size_t Columns, typename DataType>
template <size_t Rows>
Matrix<1, Columns, DataType>::Matrix(
                        Matrix<Rows, Columns, DataType> const &matrix)
{
    std::fill(d_column, d_column + Columns, DataType());

    for (size_t col = 0; col < Columns; col++)
        for (size_t row = 0; row < Rows; row++)
            d_column[col] += matrix[row][col];
}
```

Note the way the constructor's parameter is defined: it's a reference to a `Matrix` template, using the additional `Row` template parameter as well as the template parameters of the partial specialization itself.

- We don't really require additional members to satisfy our current needs. To access the data elements of the `MatrixRow` an overloaded `operator[]()` is of course useful. Again, the `const` variant can be implemented like the non-`const` variant. Here is its implementation:

```
template <size_t Columns, typename DataType>
DataType &Matrix<1, Columns, DataType>::operator[](size_t idx)
{
    return d_column[idx];
}
```

Now that we have defined the generic `Matrix` class as well as the partial specialization defining a single row, the compiler will select the row's specialization whenever a `Matrix` is defined using `Row = 1`. For example:

```
Matrix<4, 6> matrix;        // generic Matrix template is used
Matrix<1, 6> row;           // partial specialization is used
```

The partial specialization for a `MatrixColumn` is constructed similarly. Let's present its highlights (the full `Matrix` template class definition as well as all its specializations are provided in the `cplusplus.yo.zip` archive (at `fpt.rug.nl`[1]) in the file `yo/templateclasses/examples/matrix.h`):

- The template class partial specialization again starts with a template announcement. The class definition itself now specifies a fixed value for the second (generic) template parameter, illustrating that we can construct partial specializations for every single template parameter; not just the first or the last:

```
template <size_t Rows, typename DataType>
class Matrix<Rows, 1, DataType>
```

- Its constructors are implemented completely analogously to the way the `MatrixRow` constructors were implemented. Their implementations are left as an exercise to the reader (and they can be found in `matrix.h`).

- An additional member `sum()` is defined to compute the sum of the elements of a `MatrixColumn` vector. It's implementation is simply realized using the `accumulate()` generic algorithm:

```
template <size_t Rows, typename DataType>
DataType Matrix<Rows, 1, DataType>::sum()
{
    return std::accumulate(d_row, d_row + Rows, DataType());
}
```

The reader might wonder what happens if we specify the following matrix:

```
Matrix<1, 1> cell;
```

[1]ftp://ftp.rug.nl/contrib/frank/documents/annotations/

Is this a `MatrixRow` or a `MatrixColumn` specialization? The answer is: neither. It's ambiguous, precisely because *both* the columns *and* the rows could be used with a (different) template partial specialization. If such a `Matrix` is actually required, yet another specialized template must be designed. Since this template specialization can be useful to obtain the sum of the elements of a `Matrix`, it's covered here as well:

- This template class partial specialization also needs a template announcement, this time only specifying `DataType`. The class definition specifies two fixed values, using 1 for both the number of rows and the number of columns:

```
template <typename DataType>
class Matrix<1, 1, DataType>
```

- The specialization defines the usual batch of constructors. Again, constructors expecting a more generic `Matrix` type are implemented as member templates. For example:

```
template <typename DataType>
template <size_t Rows, size_t Columns>
Matrix<1, 1, DataType>::Matrix(
                        Matrix<Rows, Columns, DataType> const &matrix)
:
    d_cell(matrix.rowMarginals().sum())
{}

template <typename DataType>
template <size_t Rows>
Matrix<1, 1, DataType>::Matrix(Matrix<Rows, 1, DataType> const &matrix)
:
    d_cell(matrix.sum())
{}
```

- Since `Matrix<1, 1>` is basically a wrapper around a `DataType` value, we need members to access that latter value. A type conversion operator might be usefull, but we'll also need a `get()` member to obtain the value if the conversion operator isn't used by the compiler (which happens when the compiler is given a choice, see section 9.3). Here are the accessors (leaving out their `const` variants):

```
template <typename DataType>
Matrix<1, 1, DataType>::operator DataType &()
{
    return d_cell;
}

template <typename DataType>
DataType &Matrix<1, 1, DataType>::get()
{
    return d_cell;
}
```

The following `main()` function shows how the `Matrix` template class and its partial specializations can be used:

```
#include <iostream>
#include "matrix.h"
```

```
using namespace std;

int main(int argc, char **argv)
{
    Matrix<3, 2> matrix(cin);

    Matrix<1, 2> colMargins(matrix);
    cout << "Column marginals:\n";
    cout << colMargins[0] << " " << colMargins[1] << endl;

    Matrix<3, 1> rowMargins(matrix);
    cout << "Row marginals:\n";
    for (size_t idx = 0; idx < 3; idx++)
        cout << rowMargins[idx] << endl;

    cout << "Sum total: " << Matrix<1, 1>(matrix) << endl;
    return 0;
}
/*
    Generated output from input: 1 2 3 4 5 6

    Column marginals:
    9 12
    Row marginals:
    3
    7
    11
    Sum total: 21
*/
```

19.6 Instantiating template classes

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.

Template parameters are *also* specified when a template class defines default template parameter values, albeit that in that case the compiler will provide the defaults (cf. section 19.5 where `double` is used as the default type to be used with the template's `DataType` parameter). The actual values or types of template parameters are *never* deduced, as with template functions: to define a `Matrix` of elements that are complex values, the following construction is used:

```
Matrix<3, 5, std::complex> complexMatrix;
```

while the following construction defines a matrix of elements that are `double` values, with the compiler providing the (default) type `double`:

```
Matrix<3, 5> doubleMatrix;
```

A template class object may be *declared* using the keyword `extern`. For example, the following construction is used to *declare* the matrix `complexMatrix`:

```
extern Matrix<3, 5, std::complex> complexMatrix;
```

A template class declaration is sufficient if the compiler encounters function declarations of functions having return values or parameters which are template class objects, pointers or references. The following little source file may be compiled, although the compiler hasn't seen the definition of the `Matrix` template class. Note that generic classes as well as (partial) specializations may be declared. Furthermore, note that a function expecting or returning a template class object, reference, or parameter itself automatically becomes a template function. This is necessary to allow the compiler to tailor the function to the types of various actual arguments that may be passed to the function:

```
#include <stddef.h>

template <size_t Rows, size_t Columns, typename DataType = double>
class Matrix;

template <size_t Columns, typename DataType>
class Matrix<1, Columns, DataType>;

Matrix<1, 12> *function(Matrix<2, 18, size_t> &mat);
```

When template classes are used they have to be processed by the compiler first. So, template member functions must be known to the compiler when the template is instantiated. This does not mean that all members of a template class are instantiated when a template class object is defined. The compiler will only instantiate those members that are actually used. This is illustrated by the following simple class `Demo`, having two constructors and two members. When we create a `main()` function in which one constructor is used and one member is called, we can make a note of the sizes of the resulting object file and executable program. Next the class definition is modified such that the unused constructor and member are commented out. Again we compile and link the `main()` function and the resulting sizes are identical to the sizes obtained earlier (on my computer, using `g++` version 4.1.2) these sizes are 3904 bytes (after stripping). There are other ways to illustrate the point that only members that are used are instantiated, like using the `nm` program, showing the symbolic contents of object files. Using programs like `nm` will yield the same conclusion: *only template member functions that are actually used are initialized*. Here is an example of the template class `Demo` used for this little experiment. In `main()` only the first constructor and the first member function are called and thus only these members were instantiated:

```
#include <iostream>

template <typename Type>
class Demo
{
    Type d_data;

    public:
        Demo();
        Demo(Type const &value);

        void member1();
        void member2(Type const &value);
};

template <typename Type>
Demo<Type>::Demo()
:
    d_data(Type())
```

```
{}

template <typename Type>
void Demo<Type>::member1()
{
    d_data += d_data;
}

// the following members are commented out before compiling
// the second program

template <typename Type>
Demo<Type>::Demo(Type const &value)
:
    d_data(value)
{}

template <typename Type>
void Demo<Type>::member2(Type const &value)
{
    d_data += value;
}

int main()
{
    Demo<int> demo;
    demo.member1();
}
```

19.7 Processing template classes and instantiations

In section 18.9 the distinction between code depending on template parameters and code not depending on template parameters was introduced. The same distinction also holds true when template classes are defined and used.

Code that does not depend on template parameters is verified by the compiler 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 to the compiler when it encounters the `qsort()` function call. In practice this implies that `cstdlib` or `stdlib.h` must have been processed by the compiler before it will be able to process the template class definition.

On the other hand, if a template defines a `<typename Type>` template type parameter, which is the return type of some template member function, e.g.,

```
Type member() ...
```

then we distinguish the following situations where the compiler encounters `member()` or the class to which `member()` belongs:

- At the location in the source where template class objects are defined (called the *point of instantiation* of the template class object), the compiler will have read the template class definition, performing a basic check for syntactical correctness of member functions like `member()`. So, it won't accept a definition or declaration like `Type &&member()`, because **C++** does not support functions returning references to references. Furthermore, it will check the existence of the actual typename that is used for instantiating the object. This typename must be known to the compiler at the object's point of instantiation.

- At the location in the source where template member functions are used (which is called the template member function's point of instantiation), the `Type` parameter must of course still be known, and `member()`'s statements that depend on the `Type` template parameter are now checked for syntactical correctness. For example, if `member()` contains a statement like

  ```
  Type tmp(Type(), 15);
  ```

  then this is in principle a syntactically valid statement. However, when `Type = int` and `member()` is called, its instantiation will fail, because `int` does not have a constructor expecting two `int` arguments. Note that this is *not* a problem when the compiler instantiates an object of the class containing `member()`: at the point of instantiation of the object its `member()` member function is not instantiated, and so the invalid `int` construction remains undetected.

19.8 Declaring friends

Friend functions are normally constructed as *support* functions of a class that cannot be constructed as class members themselves. The well-known insertion operator for output streams is a case in point. Friend classes are most often seen in the context of nested classes, where the inner class declares the outer class as its friend (or the other way around). Here again we see a support mechanism: the inner class is constructed to support the outer class.

Like concrete classes, template classes may declare other functions and classes as their friends. Conversely, concrete classes may declare template classes as their friends. Here too, the friend is constructed as a special function or class augmenting or supporting the functionality of the declaring class. Although the `friend` keyword can thus be used in any type of class (concrete or template) to declare any type of function or class as a friend, when using template classes the following cases should be distinguished:

- A template class may declare a nontemplate function or class to be its friend. This is a common friend declaration, such as the insertion operator for `ostream` objects.
- A template class may declare another template function or class to be its friend. In this case, the friend's template parameters may have to be specified. If the actual values of the friend's template parameters must be equal to the template parameters of the class declaring the friend, the friend is said to be a *bound friend template* class or function. In this case the template parameters of the template in which a `friend` declaration is used determine (*bind*) the template parameters of the friend class or function, resulting in a one-to-one correspondence between the template's parameters and the friend's template parameters.
- In the most general case, a template class may declare another template function or class to be its friend, irrespective of the friend's actual template parameters. In this case an *unbound friend template* class or function is declared: the template parameters of the friend template class or function remain to be specified, and are not related in some predefined way to the template parameters of the class declaring the friend. For example, if a class has data members of various types, specified by its template parameters, and another class should be allowed direct access to these data members (so it should be a friend), we would like to specify any of the current template parameters to instantiate such a friend. Rather than specifying multiple bound friends, a single generic (unbound) friend may be declared, specifying the friend's actual template parameters only when this is required.

- The above cases, in which a template is declared as a friend, may also be encountered when concrete classes are used:
  - The concrete class declaring concrete friends has already been covered (chapter 11).
  - The equivalent of bound friends occurs if a concrete class specifies specific actual template parameters when declaring its friend.
  - The equivalent of unbound friends occurs if a concrete class declares a generic template as its friend.

19.8.1 Non-template functions or classes as friends

A template class may declare a concrete function, concrete member function or complete concrete class as its friend. Such a friend may access the template class's private members.

Concrete classes and ordinary functions can be declared as friends, but before a single class member function can be declared as a friend, the compiler must have seen the class interface declaring that member. Let's consider the various possibilities:

- A template class may declare a concrete function to be its friend. It is not completely clear *why* we would like to declare a concrete function as a friend. In ordinary cases we would like to pass an object of the class declaring the friend to the function. However, this requires us to provide the function with a template parameter without specifying its types. As the language does not support constructions like

  ```
  void function(std::vector<Type> &vector)
  ```

  unless `function()` itself is a template, it is not immediately clear how and why such a friend should be constructed. One reason, though, is to allow the function to access the class's private static members. Furthermore, such friends could themselves instantiate objects of classes declaring them as friends, and directly access such object's private members. For example:

  ```
  template <typename Type>
  class Storage
  {
      friend void basic();
      static size_t s_time;
      std::vector<Type> d_data;
      public:
          Storage();
  };

  template <typename Type>
  size_t Storage<Type>::s_time = 0;

  template <typename Type>
  Storage<Type>::Storage()
  {}

  void basic()
  {
      Storage<int>::s_time = time(0);
      Storage<double> storage;
      std::random_shuffle(storage.d_data.begin(), storage.d_data.end());
  }
  ```

- Declaring a concrete class to be a template class's friend probably has more practical implications. Here the friend-class may instantiate any kind of object of the template class, to access all of its private members thereafter. A simple forward declaration of the friend class in front of the template class definition is enough to make this work:

```
class Friend;

template <typename Type>
class Composer
{
    friend class Friend;
    std::vector<Type> d_data;
    public:
        Composer();
};

class Friend
{
    Composer<int> d_ints;
    public:
        Friend(std::istream &input);
};

inline::Friend::Friend(std::istream &input)
{
    std::copy(std::istream_iterator<int>(input),
              std::istream_iterator<int>(),
              back_inserter(d_ints.d_data));
}
```

- Alternatively, just a single member function of a concrete class may be declared as a friend. This requires that the compiler has read the friend class's interface before the friend is declared. Omitting the required destructor and overloaded assignment operators, the following shows an example of a class whose member `randomizer()` is declared as a friend of the class `Composer`:

```
template <typename Type>
class Composer;

class Friend
{
    Composer<int> *d_ints;
    public:
        Friend(std::istream &input);
        void randomizer();
};

template <typename Type>
class Composer
{
    friend void Friend::randomizer();
    std::vector<Type> d_data;
    public:
        Composer(std::istream &input)
        {
```

```
                std::copy(std::istream_iterator<int>(input),
                          std::istream_iterator<int>(),
                          back_inserter(d_data));
            }
    };

    inline Friend::Friend(std::istream &input)
    :
        d_ints(new Composer<int>(input))
    {}

    inline void Friend::randomizer()
    {
        std::random_shuffle(d_ints->d_data.begin(), d_ints->d_data.end());
    }
```

In this example note that `Friend::d_ints` is a pointer member. It cannot be a `Composer<int>` object, since the `Composer` class interface hasn't yet been seen by the compiler when it reads `Friend`'s class interface. Disregarding this and defining a data member `Composer<int> d_ints` results in the compiler generating the error

```
error: field 'd_ints' has incomplete type
```

Incomplete type, as the compiler at this points knows of the existence of the class `Composer` but as it hasn't seen `Composer`'s interface it doesn't know what size the `d_ints` data member will have.

19.8.2 Templates instantiated for specific types as friends

With *bound friend* template classes or functions there is a one-to-one mapping between the actual values of the template-friends' template parameters and the template class's template parameters declaring them as friends. In this case, the friends themselves are templates too. Here are the various possibilities:

- A template function may be declared as a friend of a template class. In this case we don't experience the problems we encountered with concrete functions declared as friends of template classes. Since the friend template function itself is a template, it may be provided with the required template parameters allowing it to specify a template class parameter. Thus we can pass an object of the class declaring the template function as its friend to the template function. The organization of the various declarations thus becomes:
  - The template class declaring the friend is itself declared;
  - The template function (to be declared as a friend) is declared;
  - The template class declaring the bound template friend function is defined;
  - The (friend) template function is defined, now having access to all the template class's (private) members.

  Note that the template friend declaration specifies its template parameters immediately following the template's function name. Without the template parameter list affixed to the function name, it would be an ordinary friend function. Here is an example showing the use of a bound friend to create a subset of the entries of a dictionary. For real life examples, a dedicated function object returning `!key1.find(key2)` is probably more useful, but for the current example, `operator==()` is acceptable:

```
template <typename Key, typename Value>
class Dictionary;

template <typename Key, typename Value>
Dictionary<Key, Value>
           subset(Key const &key, Dictionary<Key, Value> const &dict);

template <typename Key, typename Value>
class Dictionary
{
    friend Dictionary<Key, Value> subset<Key, Value>
                (Key const &key, Dictionary<Key, Value> const &dict);
    std::map<Key, Value> d_dict;
    public:
        Dictionary();
};

template <typename Key, typename Value>
Dictionary<Key, Value>
           subset(Key const &key, Dictionary<Key, Value> const &dict)
{
    Dictionary<Key, Value> ret;

    std::remove_copy_if(dict.d_dict.begin(), dict.d_dict.end(),
                        std::inserter(ret.d_dict, ret.d_dict.begin()),
                        std::bind2nd(std::equal_to<Key>(), key));
    return ret;
}
```

- By declaring a full template class as a template class's friend, all members of the friend class may access all private members of the class declaring the friend. As the friend class only needs to be declared, the organization of the declaration is much easier than when template functions are declared as friends. In the following example a class `Iterator` is declared as a friend of a class `Dictionary`. Thus, the `Iterator` is able to access `Dictionary`'s private data. There are some interesting points to note here:
  - To declare a template class as a friend, that class is simply declared as a template class before it is declared as a friend:

    ```
    template <typename Key, typename Value>
    class Iterator;

    template <typename Key, typename Value>
    class Dictionary
    {
        friend class Iterator<Key, Value>;
    ```

  - However, the friend class'ss interface may already be used, even before the compiler has seen the friend's interface:

    ```
    template <typename Key, typename Value>
    template <typename Key2, typename Value2>
    Iterator<Key2, Value2> Dictionary<Key, Value>::begin()
    {
    ```

```
        return Iterator<Key, Value>(*this);
    }
    template <typename Key, typename Value>
    template <typename Key2, typename Value2>
    Iterator<Key2, Value2> Dictionary<Key, Value>::subset(Key const &key)
    {
        return Iterator<Key, Value>(*this).subset(key);
    }
```

  - Of course, the friend's interface must still be seen by the compiler. Since it's a support class for `Dictionary`, it can safely define a `std::map` data member, which is initialized by its constructor, accessing the `Dictionary`'s private data member `d_dict`:

```
    template <typename Key, typename Value>
    class Iterator
    {
        std::map<Key, Value> &d_dict;

        public:
            Iterator(Dictionary<Key, Value> &dict)
            :
                d_dict(dict.d_dict)
            {}
```

  - The `Iterator` member `begin()` simply returns a `map` iterator. However, since it is not known to the compiler what the instantiation of the map will look like, a `map<Key, Value>::iterator` is a (deprecated) *implicit typename*. To make it an *explicit* typename, simply prefix `typename` to `begin()`'s return type:

```
    template <typename Key, typename Value>
    typename std::map<Key, Value>::iterator Iterator<Key, Value>::begin()
    {
        return d_dict.begin();
    }
```

- In the previous example we might decide that only a `Dictionary` should be able to construct an `Iterator`, as `Iterator` is closely tied to `Dictionary`. This can be realized by defining `Iterator`'s constructor in its private section, and declaring `Dictionary` `Iterator`'s friend. Consequently, only `Dictionary` can create its own `Iterator`. By declaring `Iterator`'s constructor as a *bound* friend, we ensure that it can only create `Iterators` using template parameters identical to its own. Here is how it's realized:

```
template <typename Key, typename Value>
class Iterator
{
    friend Dictionary<Key, Value>::Dictionary();

    std::map<Key, Value> &d_dict;

    Iterator(Dictionary<Key, Value> &dict);

    public:
```

  In this example, `Dictionary`'s constructor is defined as `Iterator`'s friend. Here the friend is a template member. Other members can be declared as a class's friend as well, in which case their prototypes must be used, including the types of their return values. So, assuming that

```
    std::vector<Value> sortValues()
```

is a member of `Dictionary`, returning a sorted vector of its values, then the corresponding bound friend declaration would be:

```
friend std::vector<Value> Dictionary<Key, Value>::sortValues();
```

Finally, the following basic example can be used as a prototype for situations where bound friends are useful:

```
template <typename T>           // a function
void fun(T *t)                  // template
{
    t->not_public();
};

template <typename X>           // a template class
class A
{                               // fun() is used as
                                // friend bound to A,
                                // instantiated for X,
                                // whatever X may be
    friend void fun<A<X> >(A<X> *);

    public:
        A();

    private:
        void not_public();
};

template <typename X>
A<X>::A()
{
    fun(this);
}

template <typename X>
void A<X>::not_public()
{}

int main()
{
    A<int> a;

    fun(&a);                    // fun instantiated for
                                // A<int>.
}
```

19.8.3 Unbound templates as friends

When a friend is declared as an *unbound* friend, it merely declares an existing template to be its friend, no matter how it is instantiated. This may be useful in situations where the friend should be able to instantiate objects of template classes declaring the friend, allowing the friend to access the instantiated object's private members. Again, functions, classes and member functions may be declared as unbound friends.

Here are the syntactical conventions declaring unbound friends:

- Declaring an unbound template function as a friend: any instantiation of the template function may instantiate objects of the template class and may access its private members. Assume the following template function has been defined

```
template <typename Iterator, typename Class, typename Data>
Class &ForEach(Iterator begin, Iterator end, Class &object,
                    void (Class::*member)(Data &));
```

  This template function can be declared as an unbound friend in the following template class `Vector2`:

```
template <typename Type>
class Vector2: public std::vector<std::vector<Type> >
{
    template <typename Iterator, typename Class, typename Data>
    friend Class &ForEach(Iterator begin, Iterator end, Class &object,
                void (Class::*member)(Data &));
    ...
};
```

  If the template function is defined inside some namespace, the namespace must be mentioned as well. E.g., assuming that `ForEach()` is defined in the namespace `FBB` its friend declaration becomes:

```
template <typename Iterator, typename Class, typename Data>
friend Class &FBB::ForEach(Iterator begin, Iterator end, Class &object,
                void (Class::*member)(Data &));
```

  The following example illustrates the use of an unbound friend. The class `Vector2` stores vectors of elements of template type parameter `Type`. Its `process()` member uses `ForEach()` to have its private `rows()` member called, which in turn uses `ForEach()` to call its private `columns()` member. Consequently, `Vector2` uses two instantiations of `ForEach()`, and therefore an unbound friend is appropriate here. It is assumed that `Type` class objects can be inserted into `ostream` objects (the definition of the `ForEach()` template function can be found in the `cplusplus.yo.zip` archive at the `ftp.rug.nl` ftp-server). Here is the program:

```
template <typename Type>
class Vector2: public std::vector<std::vector<Type> >
{
    typedef typename Vector2<Type>::iterator iterator;

    template <typename Iterator, typename Class, typename Data>
    friend Class &ForEach(Iterator begin, Iterator end, Class &object,
                void (Class::*member)(Data &));
    public:
        void process();

    private:
        void rows(std::vector<Type> &row);
        void columns(Type &str);
```

```
};

template <typename Type>
void Vector2<Type>::process()
{
    ForEach<iterator, Vector2<Type>, std::vector<Type> >
            (this->begin(), this->end(), *this, &Vector2<Type>::rows);
}

template <typename Type>
void Vector2<Type>::rows(std::vector<Type> &row)
{
    ForEach(row.begin(), row.end(), *this,
                                    &Vector2<Type>::columns);
    std::cout << std::endl;
}

template <typename Type>
void Vector2<Type>::columns(Type &str)
{
    std::cout << str << " ";
}

using namespace std;

int main()
{
    Vector2<string> c;
    c.push_back(vector<string>(3, "Hello"));
    c.push_back(vector<string>(2, "World"));

    c.process();
}
/*
    Generated output:

    Hello Hello Hello
    World World
*/
```

- Analogously, a full template class may be declared as a friend. This allows all instantiations of the friend's member functions to instantiate the template declaring the friend class. In this case, the class declaring the friend should offer useful functionality to different instantiations (i.e., using different arguments for its template parameters) of its friend class. The syntactical convention is comparable to the convention used when declaring an unbound friend template function:

```
template <typename Type>
class PtrVector
{
    template <typename Iterator, typename Class>
    friend class Wrapper;      // unbound friend class
};
```

All members of the template class `Wrapper` may now instantiate `PtrVectors` using any actual type for its Type template parameter, at the same time allowing Wrapper's instantiation to access all of PtrVector's private members.

- When only some members of a template class need access to the private members of another template class (e.g., the other template class has private constructors, and only some members of the first template class need to instantiate objects of the second template class), then the latter template class may declare only those members of the former template class requiring access to its private members as its friends. Again, the friend class's interface may be left unspecified. However, the compiler must be informed that the friend member's class is indeed a class. A forward declaration of that class must therefore be given as well. In the following example PtrVector declares Wrapper::begin() as its friend. Note the forward declaration of the class Wrapper:

```
template <typename Iterator>
class Wrapper;

template <typename Type>
class PtrVector
{
    template <typename Iterator> friend
        PtrVector<Type> Wrapper<Iterator>::begin(Iterator const &t1);
    ...
};
```

19.9 Template class derivation

Template classes can be used in class derivation as well. When a template class is used in class derivation, the following situations should be distinguished:

- An existing template class is used as base class when deriving a concrete class. In this case, the resulting class is still partially a template class, but this is somewhat hidden from view when an object of the derived class is constructed.
- An existing template class is used as the base class when deriving another template class. Here the template-class characteristics remain clearly visible.
- A concrete class is used as the base class when deriving a template class. This interesting hybrid allows us to construct template classes that are *partially precompiled*.

These three variants of template class derivation will now be elaborated.

Consider the following base class:

```
template<typename T>
class Base
{
    T const &t;

    public:
        Base(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);
};

template<typename T>
Derived<T>::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);
};

inline Ordinary::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.

Template class derivation pretty much follows the same rules as ordinary class derivation, not involving template classes. However, some subtleties associated with template class derivation may easily cause confusion. In the following sections class derivation involving template classes will be discussed. Some of the examples shown in these sections may contain unexpected statements and expressions, like the use of `this` when members of a template base class are called from a derived class. The 'chicken and egg' problem I encountered here was solved by first discussing the principles of template class derivation; following that discussion the subtleties that are part of template class derivation are discussed in section 19.11.

19.9.1 Deriving non-template classes from template classes

When an existing template class is used as a base class for deriving a non-template (concrete) class, the template class parameters are specified when defining the derived class's interface. If in a certain context an existing template class lacks a particular functionality, then it may be useful to derive a concrete class from a template class. For example, although the class map can easily be used in combination with the find_if() generic algorithm (section 17.4.16) to locate a particular element, it requires the construction of a class and at least two additional function objects of that class. If this is considered too much overhead in a particular context, extending a template class with some tailor-made functionality might be considered.

A program executing commands entered at the keyboard might accept all unique initial abbreviations of the commands it defines. E.g., the command list might be entered as l, li, lis or list. By deriving a class Handler from

```
map<string, void (Handler::*)(string const &cmd)>
```

and defining a process(string const &cmd) to do the actual command processing, the program might simply execute the following main() function:

```
int main()
{
    string line;
    Handler cmd;

    while (getline(cin, line))
        cmd.process(line);
}
```

The class Handler itself is derived from a complex map, in which the map's values are pointers to Handler's member functions, expecting the command line entered by the user. Here are Handler's characteristics:

- The class is derived from a std::map, expecting the command associated with each command-processing member as its keys. Since Handler uses the map merely to define associations between the commands and the processing member functions, we use private derivation here:

  ```
  class Handler: private std::map<std::string,
                          void (Handler::*)(std::string const &cmd)>
  ```

- The actual association can be defined using static private data members: s_cmds is an array of Handler::value_type values, and s_cmds_end is a constant pointer pointing beyond the array's last element:

  ```
      static value_type s_cmds[];
      static value_type *const s_cmds_end;
  ```

- The constructor simply initializes the map from these two static data members. It could be implemented inline:

  ```
  inline Handler::Handler()
  :
      std::map<std::string,
                  void (Handler::*)(std::string const &cmd)>
      (s_cmds, s_cmds_end)
  {}
  ```

- The member `process()` iterates along the map's elements. Once the first word on the command line matches the initial characters of the command, the corresponding command is executed. If no such command is found, an error message is issued:

```
void Handler::process(std::string const &line)
{
    istringstream istr(line);
    string cmd;
    istr >> cmd;
    for (iterator it = begin(); it != end(); it++)
    {
        if (it->first.find(cmd) == 0)
        {
            (this->*it->second)(line);
            return;
        }
    }
    cout << "Unknown command: " << line << endl;
}
```

19.9.2 Deriving template classes from template classes

Although it's perfectly acceptable to derive a concrete class from a template class, the resulting class of course has limited generality compared to its template base class. If generality is important, it's probably a better idea to derive a template class from a template class. This allows us the extend an existing template class with some additional functionality, like allowing hierarchical sorting of its elements.

The following class `SortVector` is a template class derived from the existing template class `Vector`. However, it allows us to perform a *hierarchical sort* of its elements, using any order of any members its data elements may contain. To accomplish this there is but one requirement: the `SortVector`'s data type must have dedicated member functions comparing its members. For example, if `SortVector`'s data type is an object of class `MultiData`, then `MultiData` should implement member functions having the following prototypes for each of its data members which can be compared:

```
bool (MultiData::*)(MultiData const &rhv)
```

So, if `MultiData` has two data members, `int d_value` and `std::string d_text`, and both may be required for a hierarchical sort, then `MultiData` should offer members like:

```
bool intCmp(MultiData const &rhv);  // returns d_value < rhv.d_value
bool textCmp(MultiData const &rhv); // returns d_text  < rhv.d_text
```

Furthermore, as a convenience it is also assumed that `operator<<()` and `operator>>()` have been defined for `MultiData` objects, but that assumption as such is irrelevant to the current discussion.

The template class `SortVector` is derived directly from the template class `std::vector`. Our implementation inherits all members from that base class, as well as two simple constructors:

```
template <typename Type>
class SortVector: public std::vector<Type>
```

```
{
    public:
        SortVector()
        {}
        SortVector(Type const *begin, Type const *end)
        :
            std::vector<Type>(begin, end)
        {}
```

However, its member `hierarchicalSort()` is the actual reason why the class exists. This class defines the hierarchical sort criteria. It expects an array of pointers to member functions of the class indicated by `sortVector`'s template `Type` parameter as well as an `size_t` indicating the size of the array. The array's first element indicates the class's most significant or first sort criterion, the array's last element indicates the class's least significant or last sort criterion. Since the `stable_sort()` generic algorithm was designed explicitly to support hierarchical sorting, the member uses this generic algorithm to sort `SortVector`'s elements. With hierarchical sorting, the least significant criterion should be sorted first. `hierarchicalSort()`'s implementation therefore, is easy, assuming the existence of a support class `SortWith` whose objects are initialized by the addresses of the member functions passed to the `hierarchicalSort()` member:

```
template <typename Type>
class SortWith
{
   bool (Type::*d_ptr)(Type const &rhv) const;
```

The class `SortWith` is a simple *wrapper class* around a pointer to a predicate function. Since it's dependent on `SortVector`'s actual data type `SortWith` itself is also a template class:

```
template <typename Type>
class SortWith
{
   bool (Type::*d_ptr)(Type const &rhv) const;
```

It's constructor receives such a pointer and initializes the class's `d_ptr` data member:

```
template <typename Type>
SortWith<Type>::SortWith(bool (Type::*ptr)(Type const &rhv) const)
:
    d_ptr(ptr)
{}
```

Its binary predicate member `operator()()` should return `true` if its first argument should be sorted before its second argument:

```
template <typename Type>
bool SortWith<Type>::operator()(Type const &lhv, Type const &rhv) const
{
    return (lhv.*d_ptr)(rhv);
}
```

Finally, an illustration is provided by the following `main()` function.

- First, A `SortVector` object is created for `MultiData` objects, using the `copy()` generic algorithm to fill the `SortVector` object from information appearing at the program's standard input stream. Having initialized the object its elements are displayed to the standard output stream:

```
SortVector<MultiData> sv;

copy(istream_iterator<MultiData>(cin),
        istream_iterator<MultiData>(),
        back_inserter(sv));
```

- An array of pointers to members is initialized with the addresses of two member functions. The text comparison is considered the most significant sort criterion:

```
bool (MultiData::*arr[])(MultiData const &rhv) const =
{
    &MultiData::textCmp,
    &MultiData::intCmp,
};
```

- Next, the array's elements are sorted and displayed to the standard output stream:

```
sv.hierarchicalSort(arr, 2);
```

- Then the two elements of the array of pointers to `MultiData`'s member functions are swapped, and the previous step is repeated:

```
swap(arr[0], arr[1]);
sv.hierarchicalSort(arr, 2);
```

After compilation the program the following command can be given:

```
echo a 1 b 2 a 2 b 1 | a.out
```

This results in the following output:

```
a 1 b 2 a 2 b 1
====
a 1 a 2 b 1 b 2
====
a 1 b 1 a 2 b 2
====
```

19.9.3 Deriving template classes from non-template classes

An existing class may be used as the base class for deriving a template class. The advantage of such an inheritance tree is that the base class's members may all be compiled beforehand, so when objects of the template class are instantiated only the used members of the derived (template) class need to be instantiated.

This approach may be used for all template classes having member functions whose implementations do not depend on template parameters. These members may be defined in a separate class which is then used as a base class of the template class derived from it.

As an illustration of this approach we'll develop such a template class in this section. We'll develop a class `Table` derived from a non-template class `TableType`. The class `Table` will display elements of some type in a table having a configurable number of columns. The elements are either displayed horizontally (the first *k* elements occupying the first row) or vertically (the first *r* elements occupying a first column).

When displaying the table's elements they are inserted into a stream. This allows us to define the handling of the table in a separate class (`TableType`), implementing the table's presentation. Since the table's elements are inserted into a stream, the conversion to text (or `string`) can be implemented in `Table`, but the handling of the strings is left to `TableType`. We'll cover some characteristics of `TableType` shortly, concentrating on `Table`'s interface first:

- The class `Table` is a template class, requiring only one template type parameter: `Iterator` refers to an iterator to some data type:

```
template <typename Iterator>
class Table: public TableType
{
```

- It requires no data members: all data manipulations are performed by `TableType`.
- It has two constructors. The constructor's first two parameters are `Iterators` used to iterate over the elements to enter into the table. Furthermore, the constructors require us to specify the number of columns we would like our table to have, as well as a *FillDirection*. `FillDirection` is an `enum` type that is actually defined by `TableType`, having values `Horizontal` and `Vertical`. To allow `Table`'s users to exercise control over headers, footers, captions, horizontal and vertical separators, one constructor has `TableSupport` reference parameter. The class `TableSupport` will be developed later as a virtual class allowing clients to exercise this control. Here are the class's constructors:

```
        Table(Iterator const &begin, Iterator const &end,
                size_t nColumns, FillDirection direction);
        Table(Iterator const &begin, Iterator const &end,
                TableSupport &tableSupport,
                size_t nColumns, FillDirection direction);
```

- The constructors are `Table`'s only two public members. Both constructors use a base class initializer to initialize their `TableType` base class and then call the class's private member `fill()` to insert data into the `TableType` base class object. Here are the constructor's implementations:

```
template <typename Iterator>
Table<Iterator>::Table(Iterator const &begin, Iterator const &end,
                TableSupport &tableSupport,
                size_t nColumns, FillDirection direction)
:
    TableType(tableSupport, nColumns, direction)
{
    fill(begin, end);
}

template <typename Iterator>
Table<Iterator>::Table(Iterator const &begin, Iterator const &end,
                size_t nColumns, FillDirection direction)
:
```

```
    TableType(nColumns, direction)
{
    fill(begin, end);
}
```

- The class's `fill()` member iterates over the range of elements `[begin, end)`, as defined by the constructor's first two parameters. As we will see shortly, `TableType` defines a protected data member `std::vector<std::string> d_string`. One of the requirements of the data type to which the iterators point is that this data type can be inserted into streams. So, `fill()` uses a `ostringstream` object to obtain the textual representation of the data, which is then appended to `d_string`:

```
template <typename Iterator>
void Table<Iterator>::fill(Iterator it, Iterator const &end)
{
    while (it != end)
    {
        std::ostringstream str;
        str << *it++;
        d_string.push_back(str.str());
    }
    init();
}
```

This completes the implementation of the class `Table`. Note that this template class only has three members, two of them constructors. Therefore, in most cases only two template functions will have to be instantiated: a constructor and the class's `fill()` member. For example, the following constructs a table having four columns, vertically filled by `string`s extracted from the standard input stream:

```
Table<istream_iterator<string> >
    table(istream_iterator<string>(cin), istream_iterator<string>(),
          4, TableType::Vertical);
```

Note here that the fill-direction is specified as `TableType::Vertical`. It could also have been specified using `Table`, but since `Table` is a template class, the specification would become somewhat more complex: `Table<istream_iterator<string> >::Vertical`.

Now that the `Table` derived class has been designed, let's turn our attention to the class `TableType`. Here are its essential characteristics:

- It is a concrete class, designed to operate as `Table`'s base class.
- It uses various private data members, among which `d_colWidth`, a vector storing the width of the widest element per column and `d_indexFun`, pointing to the class's member function returning the element in `table[row][column]`, conditional to the table's fill direction. `TableType` also uses a `TableSupport` pointer and a reference. The constructor not requiring a `TableSupport` object uses the `TableSupport *` to allocate a (default) `TableSupport` object and then uses the `TableSupport &` as the object's alias. The other constructor initializes the pointer to 0, and uses the reference data member to refer to the `TableSupport` object provided by its parameter. Alternatively, a static `TableSupport` object might have been used to initialize the reference data member in the former constructor. The remaining private data members are probably self-explanatory:

```
        TableSupport           *d_tableSupportPtr;
```

```
        TableSupport           &d_tableSupport;
        size_t                 d_maxWidth;
        size_t                 d_nRows;
        size_t                 d_nColumns;
        WidthType                d_widthType;
        std::vector<size_t>    d_colWidth;
        size_t                (TableType::*d_widthFun)
                                          (size_t col) const;
        std::string const      &(TableType::*d_indexFun)
                                          (size_t row, size_t col) const;
```

- The actual `string` objects populating the table are stored in a protected data member:

```
        std::vector<std::string> d_string;
```

- The (protected) constructors perform basic tasks: they initialize the object's data members. Here is the constructor expecting a reference to a `TableSupport` object:

```
#include "tabletype.ih"

TableType::TableType(TableSupport &tableSupport, size_t nColumns,
                        FillDirection direction)
:
    d_tableSupportPtr(0),
    d_tableSupport(tableSupport),
    d_maxWidth(0),
    d_nRows(0),
    d_nColumns(nColumns),
    d_widthType(ColumnWidth),
    d_colWidth(nColumns),
    d_widthFun(&TableType::columnWidth),
    d_indexFun(direction == Horizontal ?
                    &TableType::hIndex
                :
                    &TableType::vIndex)
{}
```

- Once `d_string` has been filled, the table is initialized by `Table::fill()`. The `init()` protected member resizes `d_string` so that its size is exactly `rows x columns`, and it determines the maximum width of the elements per column. Its implementation is straightforward:

```
#include "tabletype.ih"

void TableType::init()
{
    if (!d_string.size())                     // no elements
        return;                               // then do nothing

    d_nRows = (d_string.size() + d_nColumns - 1) / d_nColumns;
    d_string.resize(d_nRows * d_nColumns);  // enforce complete table

                                              // determine max width per column,
                                              // and max column width
    for (size_t col = 0; col < d_nColumns; col++)
    {
```

```
        size_t width = 0;
        for (size_t row = 0; row < d_nRows; row++)
        {
            size_t len = stringAt(row, col).length();
            if (width < len)
                width = len;
        }
        d_colWidth[col] = width;

        if (d_maxWidth < width)             // max. width so far.
            d_maxWidth = width;
    }
}
```

- The public member `insert()` is used by the insertion operator (`operator<<()`) to insert a `Table` into a stream. First it informs the `TableSupport` object about the table's dimensions. Next it displays the table, allowing the `TableSupport` object to write headers, footers and separators:

```
#include "tabletype.ih"

ostream &TableType::insert(ostream &ostr) const
{
    if (!d_nRows)
        return ostr;

    d_tableSupport.setParam(ostr, d_nRows, d_colWidth,
                            d_widthType == EqualWidth ? d_maxWidth : 0);

    for (size_t row = 0; row < d_nRows; row++)
    {
        d_tableSupport.hline(row);

        for (size_t col = 0; col < d_nColumns; col++)
        {
            size_t colwidth = width(col);

            d_tableSupport.vline(col);
            ostr << setw(colwidth) << stringAt(row, col);
        }

        d_tableSupport.vline();
    }
    d_tableSupport.hline();

    return ostr;
}
```

- The `cplusplus.yo.zip` archive contains `TableSupport`'s full implementation. This implementation is found in the directory `yo/templateclasses/examples/table`. Most of its remaining members are private. Among those, the following two members return table element `[row][column]` for, respectively, a horizontally filled table and a vertically filled table:

```
    inline std::string const &TableType::hIndex(size_t row, size_t col) const
    {
```

```
        return d_string[row * d_nColumns + col];
    }
    inline std::string const &TableType::vIndex(size_t row, size_t col) const
    {
        return d_string[col * d_nRows + row];
    }
```

The support class `TableSupport` is used to display headers, footers, captions and separators. It has four virtual members to perform those tasks (and, of course, a virtual constructor):

- `hline(size_t rowIndex)`: called just before the elements in row `rowIndex` will be displayed.
- `hline()`: called immediately after displaying the final row.
- `vline(size_t colIndex)`: called just before the element in column `colIndex` will be displayed.
- `vline()`: called immediately after displaying all elements in a row.

The reader is referrred to the `cplusplus.yo.zip` archive for the full implementation of the classes `Table`, `TableType` and `TableSupport`. Here is a small program showing their use:

```
/*
                              table.cc
*/

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

#include "tablesupport/tablesupport.h"
#include "table/table.h"

using namespace std;
using namespace FBB;

int main(int argc, char **argv)
{
    size_t nCols = 5;
    if (argc > 1)
    {
        istringstream iss(argv[1]);
        iss >> nCols;
    }

    istream_iterator<string>   iter(cin);   // first iterator isn't const

    Table<istream_iterator<string> >
        table(iter, istream_iterator<string>(), nCols,
              argc == 2 ? TableType::Vertical : TableType::Horizontal);
```

```
        cout << table << endl;
        return 0;
    }
    /*
        Example of generated output:
        After: echo a b c d e f g h i j | demo 3
            a e i
            b f j
            c g
            d h
        After: echo a b c d e f g h i j | demo 3 h
            a b c
            d e f
            g h i
            j
    */
```

19.10 Template classes and nesting

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 a class `PtrVector`, a class `iterator` is defined. The nested class receives its information from its surrounding class, a `PtrVector<Type>` class. Since this surrounding class should be the only class constructing its iterators, `iterator`'s constructor is made private, and the surrounding class is given access to the private members of `iterator` using a *bound friend* declaration. Here is the initial section of `PtrVector`'s class interface:

```
    template <typename Type>
    class PtrVector: public std::vector<Type *>
```

This shows that the `std::vector` base class will store pointers to `Type` values, rather than the values themselves. Of course, a destructor is needed now, since the (externally allocated) memory for the `Type` objects must eventually be freed. Alternatively, the allocation might be part of `PtrVector`'s tasks, when storing new elements. Here it is assumed that the `PtrVector`'s clients do the required allocations, and that the destructor will be implemented later on.

The nested class defines its constructor as a private member, and allows `PtrVector<Type>` objects to access its private members. Therefore only objects of the surrounding `PtrVector<Type>` class type are allowed to construct their `iterator` objects. However, `PtrVector<Type>`'s clients may construct *copies* of the `PtrVector<Type>::iterator` objects they use. Here is the nested class `iterator`, containing the required `friend` declaration. Note the use of the `typename` keyword: since `std::vector<Type *>::iterator` depends on a template parameter, it is not yet an instantiated class, so `iterator` becomes an implicit typename. The compiler issues a corresponding warning if `typename` has been omitted. In these cases `typename` must be used. Here is the class interface:

```
            class iterator
            {
                friend class PtrVector<Type>;
                typename std::vector<Type *>::iterator d_begin;

                iterator(PtrVector<Type> &vector);
```

```
            public:
                Type &operator*();
        };
```

The implementation of the members shows that the base class's `begin()` member is called to initialize `d_begin`. Also note that the return type of `PtrVector<Type>::begin()` must again be preceded by `typename`:

```
template <typename Type>
PtrVector<Type>::iterator::iterator(PtrVector<Type> &vector)
:
    d_begin(vector.std::vector<Type *>::begin())
{}

template <typename Type>
Type &PtrVector<Type>::iterator::operator*()
{
    return **d_begin;
}
```

The remainder of the class is simple. Omitting all other functions that might be implemented, the function `begin()` will return a newly constructed `PtrVector<Type>::iterator` object. It may call the constructor since the class `iterator` called its surrounding class its friend:

```
template <typename Type>
typename PtrVector<Type>::iterator PtrVector<Type>::begin()
{
    return iterator(*this);
}
```

Here is a simple skeleton program, showing how the nested class `iterator` might be used:

```
int main()
{
    PtrVector<int> vi;

    vi.push_back(new int(1234));

    PtrVector<int>::iterator begin = vi.begin();

    std::cout << *begin << endl;
}
```

Nested enumerations and typedefs can also be defined in template classes. The class `Table`, mentioned before (section 19.9.3) inherited the enumeration `TableType::FillDirection`. If `Table` would have been implemented as a full template class, then this enumeration would have been defined in `Table` itself as:

```
template <typename Iterator>
class Table: public TableType
{
```

```
        public:
            enum FillDirection
            {
                Horizontal,
                Vertical
            };
        ...
    };
```

In this case, the actual value of the template type parameter must be specified when referring to a `FillDirection` value or to its type. For example (assuming `iter` and `nCols` are defined as in section 19.9.3):

```
    Table<istream_iterator<string> >::FillDirection direction =
                argc == 2 ?
                    Table<istream_iterator<string> >::Vertical
                :
                    Table<istream_iterator<string> >::Horizontal;

    Table<istream_iterator<string> >
        table(iter, istream_iterator<string>(), nCols, direction);
```

19.11 Subtleties with template classes

19.11.1 Type resolution for base class members

Consider the following example of a template base and a derived class:

```
    #include <iostream>

    template <typename T>
    class Base
    {
        public:
            void member();
    };

    template <typename T>
    void Base<T>::member()
    {
        std::cout << "This is Base<T>::member()\n";
    }

    template <typename T>
    class Derived: public Base<T>
    {
        public:
            Derived();
    };

    template <typename T>
```

```
Derived<T>::Derived()
{
    member();
}
```

This example won't compile, and the compiler tells us something like:

```
error: there are no arguments to 'member' that depend on a template
       parameter, so a declaration of 'member' must be available
```

At first glance, this error may cause some confusion, since with non-template classes public and protected base class members are immediately available. This holds also true for template classes, but only if the compiler can figure out what we mean. In the above situation, the compiler can't, since it doesn't know for what type `T` the member function `member` must be initialized.

To appreciate why this is true, consider the situation where we have defined a specialization:

```
template <>
Base<int>::member()
{
    std::cout << "This is the int-specialization\n";
}
```

Since the compiler, when processing the class `Derived`, can't be sure that no specialization will be in effect once an instantiation of `Derived` is called for, it can't decide yet for what type to instantiate `member`, since `member()`'s call in `Derived::Derived()` doesn't require a template type parameter. In cases like these, where no template type parameter is available to determine which type to use, the compiler must be told that it should postpone its decision about the template type parameter to use for `member()` until instantiation time. This can be realized in two ways: either by using `this`, or by explicitly mentioning the base class, instantiated for the derived class's template type(s). In the following `main()` function both forms are used. Note that with the `int` template type the `int` specialization is used.

```
#include <iostream>

template <typename T>
class Base
{
    public:
        void member();
};

    template <typename T>
    void Base<T>::member()
    {
        std::cout << "This is Base<T>::member()\n";
    }

    template <>
    void Base<int>::member()
    {
        std::cout << "This is the int-specialization\n";
    }
```

```
template <typename T>
class Derived: public Base<T>
{
    public:
        Derived();
};

    template <typename T>
    Derived<T>::Derived()
    {
        this->member();
        Base<T>::member();
    }

int main()
{
    Derived<double> d;
    Derived<int> i;
}

/*
    Generated output:
    This is Base<T>::member()
    This is Base<T>::member()
    This is the int-specialization
    This is the int-specialization
*/
```

19.11.2 Returning types nested under template classes

In section 19.1.3 the keyword `typename` was introduced to allow the compiler to distinguish between template class members and types that are defined within template classes. The `typename` keyword allows us to tell the compiler that we have a type in mind that is nested under a template class.

Consider the following example in which a nested class, that is not depending on a template parameter, is defined within a template class. Furthermore, the template class member `nested()` should return an object of the nested class. Note that in this example the (deprecated) member implementation inside the class interface is used:

```
template <typename T>
class Outer
{
    public:
        class Nested
        {
        };

    Nested nested() const
    {
        return Nested();
    }
};
```

The above example compiles flawlessly: within the class `Outer` there is no ambiguity with respect to the meaning of `nested()`'s return type. Since it is advised to implement inline and template members below their class interface (see section 6.3.1), we now remove the implementation from the interface itself, and put it below the interface. Suddenly the compiler refuses to compile our member `nested()`:

```
template <typename T>
class Outer
{
    public:
        class Nested
        {
        };

    Nested nested() const;
};

template <typename T>
Outer<T>::Nested Outer<T>::nested() const
{
    return Nested();
}
```

The above implementation of `nested()` produces an error message like

error: expected constructor, destructor, or type conversion before 'Outer'.

In this case a type specification is required as `Outer<T>::Nested` refers to a *type*, nested under `Outer<T>` rather than to a member of `Outer<T>`. In situations like these, where a type that is defined as a *nested type* in a template class is returned, the `typename` keyword must be used to coerce the compiler into interpreting `Outer<T>::Nested` as a type name. Writing `typename` in front of `Outer<T>::Nested` removes the compilation error and the correct implementation of the function `nested()` becomes:

```
template <typename T>
typename Outer<T>::Nested Outer<T>::nested() const
{
    return Nested();
}
```

19.12 Constructing iterators

In section 17.2 the iterators used with generic algorithms were introduced. We've seen that several types of iterators were distinguished: InputIterators, ForwardIterators, OutputIterators, BidirectionalIterators and RandomAccessIterators.

In section 17.2 the characteristics of iterators were introduced: all iterators should support an increment operation, a dereference operation and a comparison for (in)equality.

However, when iterators must be used in the context of generic algorithms they must meet additional 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 an iterator-object is used it must be able to specify the kind of iterator it represents.

To ensure that an object of a class is interpreted as a particular type of iterator, the class must be derived from the `class iterator`. The particular type of iterator is defined by the template class's *first* parameter, and the particular data type to which the iterator points is defined by the template class's *second* parameter. Before a class may be inherited from the class `iterator`, the following header file must have been included:

```
#include <iterator>
```

The particular type of iterator that is implemented by the derived class is specified using a so-called *iterator_tag*, provided as the first template argument of the class `iterator`. For the five basic iterator types, these tags are:

- `std::input_iterator_tag`. This tag defines an InputIterator. Iterators of this type allow reading operations, iterating from the first to the last element of the series to which the iterator refers.
- `std::output_iterator_tag`. This tag defines an OutputIterator. Iterators of this type allow for assignment operations, iterating from the first to the last element of the series to which the iterator refers.
- `std::forward_iterator_tag`. This tag defines a `ForwardIterator`. Iterators of this type allow reading *and* assignment operations, iterating from the first to the last element of the series to which the iterator refers.
- `std::bidirectional_iterator_tag`. This tag defines a BidirectionalIterator. Iterators of this type allow reading *and* assignment operations, iterating step by step, possibly in alternating directions, over all elements of the series to which the iterator refers.
- `std::random_access_iterator_tag`. This tag defines a RandomAccessIterator. Iterators of this type allow reading *and* assignment operations, iterating, possibly in alternating directions, over all elements of the series to which the iterator refers, using any available (random) stepsize.

Each *iterator tag* assumes that a certain set of operators is available. The *RandomAccessIterator* is the most complex of iterators, as it implies all other iterators.

Note that iterators are always defined over a certain range, e.g., `[begin, end)`. Increment and decrement operations may result in undefined behavior of the iterator if the resulting iterator value would refer to a location outside of this range.

Often, iterators only access the elements of the series to which they refer. Internally, an iterator may use an ordinary pointer, but it is hardly ever necessary for the iterator to allocate its own memory. Therefore, as the overloaded assignment operator and the copy constructor do not have to allocate any memory, the *default implementation* of the overloaded assignment operator and of the copy constructor is usually sufficient. I.e., usually these members do not have to be implemented at all. As a consequence there is usually also no *destructor*.

Most classes offering members returning iterators do so by having members constructing the required iterator, which is thereupon returned as an object by these member functions. As the *caller* of these member functions only has to *use* or sometimes *copy* the returned iterator objects, there is normally no need to provide any publicly available constructors, except for the copy constructor. Therefore these constructors may usually be defined as *private* or *protected* members. To allow an outer class to create iterator objects, the iterator class will declare the outer class as its *friend*.

In the following sections, the construction of a *RandomAccessIterator*, the most complex of all iterators, and the construction of a *reverse RandomAccessIterator* is discussed. The container class for which a random access iterator must be developed may actually store its data elements in many different ways, e.g., using various containers or using pointers to pointers. Therefore it is difficult to construct a template iterator class which is suitable for a large variety of concrete (container) classes.

In the following sections, the available `std::iterator` class will be used to construct an inner class representing a random access iterator. This approach clearly shows how to construct an iterator class. The reader may either follow this approach when constructing iterator classes in other contexts, or a full template iterator class can be designed. An example of such a template iterator class is provided in section 20.5.

The construction of the random access iterator as shown in the next sections aims at the realization of an iterator reaching the elements of a series of elements only accessible through pointers. The iterator class is designed as an inner class of a class derived from a vector of string pointers.

19.12.1 Implementing a 'RandomAccessIterator'

When discussing containers (chapter 12) it was noted that containers own the information they contain. If they contain objects, then these objects are destroyed once the containers are destroyed. As pointers are no objects, and as `auto_ptr` objects cannot be stored in containers, using pointer data types for containers was discouraged. However, we might be able to use pointer data types in specific contexts. In the following class `StringPtr`, a concrete class is derived from the `std::vector` container using `std::string *` as its data type:

```
#ifndef _INCLUDED_STRINGPTR_H_
#define _INCLUDED_STRINGPTR_H_

#include <string>
#include <vector>

class StringPtr: public std::vector<std::string *>
{
    public:
        StringPtr(StringPtr const &other);
        ~StringPtr();

        StringPtr &operator=(StringPtr const &other);
};

#endif
```

Note the declaration of the destructor: as the object stores string pointers, a destructor is required to destroy the strings when the `StringPtr` object itself is destroyed. Similarly, a copy constructor and overloaded assignment is required. Other members (in particular: constructors) are not explicitly declared as they are not relevant to this section's topic.

Let's assume that we want to be able to use the `sort()` generic algorithm with `StringPtr` objects. This algorithm (see section 17.4.58) requires two *RandomAccessIterators*. Although these iterators are available (via `std::vector`'s `begin()` and `end()` members), they return iterators to `std::string *`s, which cannot sensibly be compared.

To remedy this, assume that we have defined an internal type `StringPtr::iterator`, not returning iterators to pointers, but iterators to the objects these pointers point to. Once this `iterator` type is available, we can add the following members to our `StringPtr` class interface, hiding the identically named, but useless members of its base class:

```
StringPtr::iterator begin();    // returns iterator to the first element
StringPtr::iterator end();      // returns iterator beyond the last
                                // element
```

Since these two members return the (proper) iterators, the elements in a `StringPtr` object can easily be sorted:

```
in main()
{
    StringPtr sp;               // assume sp is somehow filled

    sort(sp.begin(), sp.end()); // sp is now sorted
    return 0;
}
```

To make this all work, the type `StringPtr::iterator` must be defined. As suggested by its type name, `iterator` is a nested type of `StringPtr`, suggesting that we may implement `iterator` as a nested class of `StringPtr`. However, to use a `StringPtr::iterator` in combination with the `sort()` generic algorithm, it must also be a `RandomAccessIterator`. Therefore, `StringPtr::iterator` itself must be derived from the existing class `std::iterator`, available once the following preprocessor directive has been specified:

```
        #include <iterator>
```

To derive a class from `std::iterator`, both the iterator type and the data type the iterator points to must be specified. Take caution: our iterator will take care of the `string *` dereferencing; so the required data type will be `std::string`, and *not* `std::string *`. So, the class `iterator` starts its interface as:

```
class iterator:
    public std::iterator<std::random_access_iterator_tag, std::string>
```

Since its base class specification is quite complex, we could consider associating this type with a shorter name, using the following `typedef`:

```
typedef std::iterator<std::random_access_iterator_tag, std::string>
        Iterator;
```

However, if the defined type (`Iterator`) is used only once or twice, the typedefinition only adds clutter to the interface, and is better not used.

Now we're ready to redesign `StringPtr`'s class interface. It contains members returning (reverse) iterators, and a nested `iterator` class. The members will be discussed in some detail next:

```
class StringPtr: public std::vector<std::string *>
{
    public:
    class iterator: public
```

```
            std::iterator<std::random_access_iterator_tag, std::string>
    {
        friend class StringPtr;
        std::vector<std::string *>::iterator d_current;

        iterator(std::vector<std::string *>::iterator const &current);

        public:
            iterator &operator--();
            iterator const operator--(int);
            iterator &operator++();
            bool operator==(iterator const &other) const;
            bool operator!=(iterator const &other) const;
            int operator-(iterator const &rhs) const;
            std::string &operator*() const;
            bool operator<(iterator const &other) const;
            iterator const operator+(int step) const;
            iterator const operator-(int step) const;
            iterator &operator+=(int step); // increment over ‘n’ steps
            iterator &operator-=(int step); // decrement over ‘n’ steps
            std::string *operator->() const;// access the fields of the
                                            // struct an iterator points
                                            // to. E.g., it->length()
    };

    typedef std::reverse_iterator<iterator> reverse_iterator;

    iterator begin();
    iterator end();
    reverse_iterator rbegin();
    reverse_iterator rend();
};
```

Let's first have a look at `StringPtr::iterator`'s characteristics:

- `iterator` defines `StringPtr` as its friend, so `iterator`'s constructor can remain private: only the `StringPtr` class itself is now able to construct `iterators`, which seems like a sensible thing to do. Under the current implementation, *copy-constructing* remains of course possible. Furthermore, since an iterator is already provided by `StringPtr`'s base class, we can use that iterator to access the information stored in the `StringPtr` object.
- `StringPtr::begin()` and `StringPtr::end()` may simply return `iterator` objects. Their implementations are:

  ```
  inline StringPtr::iterator StringPtr::begin()
  {
      return iterator(this->std::vector<std::string *>::begin());
  }
  inline StringPtr::iterator StringPtr::end()
  {
      return iterator(this->std::vector<std::string *>::end());
  }
  ```

- All of `iterator`'s remaining members are public. It's very easy to implement them, mainly manipulating and dereferencing the available iterator `d_current`. A `RandomAccessIterator` (which is the most complex of iterators) requires a series of operators. They usually have very simple implementations, making them good candidates for inline-members:

- `iterator &operator++()`: the pre-increment operator:

  ```
  inline StringPtr::iterator &StringPtr::iterator::operator++()
  {
      ++d_current;
      return *this;
  }
  ```

- `iterator &operator--()`: the pre-decrement operator:

  ```
  inline StringPtr::iterator &StringPtr::iterator::operator--()
  {
      --d_current;
      return *this;
  }
  ```

- `iterator operator--()`: the post-decrement operator:

  ```
  inline StringPtr::iterator const StringPtr::iterator::operator--(int)
  {
      return iterator(d_current--);
  }
  ```

- `iterator &operator=(iterator const &other)`: the overloaded assignment operator. Since `iterator` objects do not allocate any memory themselves, the default assignment operator will do.

- `bool operator==(iterator const &rhv) const`: testing the equality of two `iterator` objects:

  ```
  inline bool StringPtr::iterator::operator==(iterator const &other) const
  {
      return d_current == other.d_current;
  }
  ```

- `bool operator<(iterator const &rhv) const`: tests whether the left-hand side iterator points to an element of the series located *before* the element pointed to by the right-hand side iterator:

  ```
  inline bool StringPtr::iterator::operator<(iterator const &other) const
  {
      return **d_current < **other.d_current;
  }
  ```

- `int operator-(iterator const &rhv) const`: returns the number of elements between the element pointed to by the left-hand side iterator and the right-hand side iterator (i.e., the value to add to the left-hand side iterator to make it equal to the value of the right-hand side iterator):

  ```
  inline int StringPtr::iterator::operator-(iterator const &rhs) const
  {
      return d_current - rhs.d_current;
  }
  ```

- `Type &operator*() const`: returns a reference to the object to which the current iterator points. With an `InputIterator` and with all `const_iterators`, the return type of this overloaded operator should be `Type const &`. This operator returns a reference to a string. This string is obtained by dereferencing the dereferenced `d_current` value. As d_current is an iterator to string * elements, two dereference operations are required to reach the string itself:

```
inline std::string &StringPtr::iterator::operator*() const
{
    return **d_current;
}
```

– iterator const operator+(int stepsize) const: this operator advances the current iterator by stepsize steps:

```
inline StringPtr::iterator const
        StringPtr::iterator::operator+(int step) const
{
    return iterator(d_current + step);
}
```

– iterator const operator-(int stepsize) const: this operator decreases the current iterator by stepsize steps:

```
inline StringPtr::iterator const
       StringPtr::iterator::operator-(int step) const
{
    return iterator(d_current - step);
}
```

– iterators may be constructed from existing iterators. This constructor doesn't have to be implemented, as the default copy constructor can be used.

– std::string *operator->() const is an additionally added operator. Here only one dereference operation is required, returning a pointer to the string, allowing us to access the members of a string via its pointer.

```
inline std::string *StringPtr::iterator::operator->() const
{
    return *d_current;
}
```

– Two more additionally added operators are operator+=() and operator-=(). They are not formally required by RandomAccessIterators, but they come in handy anyway:

```
inline StringPtr::iterator &StringPtr::iterator::operator+=(int step)
{
    d_current += step;
    return *this;
}
inline StringPtr::iterator &StringPtr::iterator::operator-=(int step)
{
    d_current -= step;
    return *this;
}
```

The interfaces required for other iterator types are simpler, requiring only a subset of the interface required by a random access iterator. E.g., the forward iterator is never decremented and never incremented over arbitrary step sizes. Consequently, in that case all decrement operators and operator+(int step) can be omitted from the interface. Of course, the tag to use would then be std::forward_iterator_tag. The tags (and the set of required operators) varies accordingly for the other iterator types.

19.12.2 Implementing a 'reverse_iterator'

Once we've implemented an iterator, the matching *reverse iterator* can be implemented in a jiffy. Comparable to the `std::iterator` a `std::reverse_iterator` exists, which will nicely implement the reverse iterator for us, once we have defined an iterator class. Its constructor merely requires an object of the iterator type for which we want to construct a reverse iterator.

To implement a reverse iterator for `StringPtr`, we only need to define the `reverse_iterator` type in its interface. This requires us to specify only one line of code, which must be inserted after the interface of the class `iterator`:

```
        typedef std::reverse_iterator<iterator> reverse_iterator;
```

Finally, the well known members `rbegin()` and `rend()` are added to `StringPtr`'s interface. Again, they can easily be implemented inline:

```
inline StringPtr::reverse_iterator StringPtr::rbegin()
{
    return reverse_iterator(end());
}
inline StringPtr::reverse_iterator StringPtr::rend()
{
    return reverse_iterator(begin());
}
```

Note the arguments the `reverse_iterator` constructors receive: the *begin point* of the reversed iterator is obtained by providing `reverse_iterator`'s constructor with `end()`: the *endpoint* of the normal iterator range; the *endpoint* of the reversed iterator is obtained by providing `reverse_iterator`'s constructor with `begin()`: the *begin point* of the normal iterator range.

The following little program illustrates the use of `StringPtr`'s `RandomAccessIterator`:

```
    #include <iostream>
    #include <algorithm>
    #include "stringptr.h"
    using namespace std;

    int main(int argc, char **argv)
    {
        StringPtr sp;

        while (*argv)
            sp.push_back(new string(*argv++));

        sort(sp.begin(), sp.end());
        copy(sp.begin(), sp.end(), ostream_iterator<string>(cout, " "));

        cout << "\n======\n";

        sort(sp.rbegin(), sp.rend());
        copy(sp.begin(), sp.end(), ostream_iterator<string>(cout, " "));

        cout << endl;
```

```
}
/*
        when called as:
    a.out bravo mike charlie zulu quebec

        generated output:
    a.out bravo charlie mike quebec zulu
    ======
    zulu quebec mike charlie bravo a.out
*/
```

Although it is thus possible to construct a reverse iterator from a normal iterator, the opposite does not hold true: it is not possible to initialize a normal iterator from a reverse iterator. Let's assume we would like to process all lines stored in a `vector<string> lines` up to any trailing empty lines (or lines only containing blanks) it might contain. How would we proceed? One approach is to start the processing from the first line in the vector, continuing until the first of the trailing empty lines. However, once we encounter an empty line it does of course not have to be the first line of the set of trailing empty lines. In that case, we would like to use the following algorithm:

- First, use

  ```
  rit = find_if(lines.rbegin(), lines.rend(), NonEmpty());
  ```

  to obtain a `reverse_iterator rit` pointing to the last non-empty line.

- Next, use

  ```
  for_each(lines.begin(), --rit, Process());
  ```

  to process all lines up to the first empty line.

However, we can't mix iterators and reverse iterators when using generic algorithms. So how can we initialize the second iterator using the available `reverse_iterator`? The solution is actually not very difficult, as an iterator may be initialized by a pointer. The reverse iterator `rit` is not a pointer, but `&*(rit - 1)` or `&*-rit` is. Thus, we can use

```
for_each(lines.begin(), &*--rit, Process());
```

to process all the lines up to the first of the set of trailing empty lines. In general, if `rit` is a `reverse_iterator` pointing to some element, but we need an `iterator` to point to that element, we may use `&*rit` to initialize the iterator. Here, the dereference operator is applied to reach the element the reverse iterator refers to. Then the address operator is applied to obtain its address.

Chapter 20

Concrete examples of C++

In this chapter several concrete examples of **C++** programs, classes and templates will be presented. Topics covered by this document such as virtual functions, `static` members, etc. are illustrated in this chapter. The examples roughly follow the organization of earlier chapters.

First, examples using `stream` classes are presented, including some detailed examples illustrating polymorphism. 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.

Next, several templates will be developed, both template functions and full template classes.

Finally, we'll touch the subjects of scanner and parser generators, and show how these tools may be used in **C++** programs. These final examples assume a certain familiarity with the concepts underlying these tools, like grammars, parse-trees and parse-tree decoration. Once the input for a program exceeds a certain level of complexity, it's advantageous to use scanner- and parser-generators to produce code doing the actual input recognition. One of the examples in this chapter describes the usage of these tools in a **C++** environment.

20.1 Using file descriptors with 'streambuf' classes

20.1.1 Classes for output operations

Extensions to the ANSI/ISO standard may be available allowing us to read from and/or write to *file descriptors*. However, such extensions are not standard, and may thus vary or be unavailable across compilers and/or compiler versions. On the other hand, a file descriptor can be considered a device. So it seems natural to use the class `streambuf` as the starting point for constructing classes interfacing file descriptors.

In this section we will construct classes which may be used to write to a device identified by a file descriptor: it may be a file, but it could also be a pipe or socket. Section 20.1.2 discusses reading from devices given their file descriptors, while section 20.3.1 reconsiders redirection, discussed earlier in section 5.8.3.

Basically, deriving a class for output operations is simple. The only member function that *must* be overridden is the `virtual` member `int overflow(int c)`. This member is responsible for writing characters to the device once the class's buffer is full. If `fd` is a file descriptor to which information may be written, and if we decide against using a buffer then the member `overflow()` can simply be:

```
class UnbufferedFD: public std::streambuf
{
    public:
        int overflow(int c);
        ...
};

int UnbufferedFD::overflow(int c)
{
    if (c != EOF)
    {
        if (write(d_fd, &c, 1) != 1)
            return EOF;
    }
    return c;
}
```

The argument received by `overflow()` is either written as a value of type `char` to the file descriptor, or `EOF` is returned.

This simple function does not use an output buffer. As the use of a buffer is strongly advised (see also the next section), the construction of a class using an output buffer will be discussed next in somewhat greater detail.

When an output buffer is used, the `overflow()` member will be a bit more complex, as it is now only called when the buffer is full. Once the buffer is full, we *first* have to flush the buffer, for which the (virtual) function `streambuf::sync()` is available. Since `sync()` is a virtual function, classes derived from `std::streambuf` may redefine `sync()` to flush a buffer `std::streambuf` itself doesn't know about.

Overriding `sync()` and using it in `overflow()` is not all that has to be done: eventually we might have less information than fits into the buffer. So, at the end of the lifetime of our special `streambuf` object, its buffer might only be partially full. Therefore, we must make sure that the buffer is flushed once our object goes out of scope. This is of course very simple: `sync()` should be called by 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 will add a couple of additional features, though.

- First, we should allow the user of the class to specify the size of the output buffer.
- Second, it should be possible to construct an object of our class before the file descriptor is actually known. Later, in section 20.3 we'll encounter a situation where this feature will be used.

In order to save some space, the successful operation of the various functions was not checked. In 'real life' implementations these checks should of course not be omitted. Our class `ofdnstreambuf` has the following characteristics:

- The class itself is derived from `std::streambuf`. It defines three data members, keeping track of the size of the buffer, the file descriptor and the buffer itself. Here is the full class interface

```
class ofdnstreambuf: public std::streambuf
{
    size_t d_bufsize;
    int    d_fd;
    char   *d_buffer;

    public:
        ofdnstreambuf();
        ofdnstreambuf(int fd, size_t bufsize = 1);
        ~ofdnstreambuf();
        void open(int fd, size_t bufsize = 1);
        int sync();
        int overflow(int c);
};
```

- Its default constructor merely initializes the buffer to 0. Slightly more interesting is its constructor expecting a filedescriptor and a buffer size: it simply passes its arguments on to the class's `open()` member (see below). Here are the constructors:

```
inline ofdnstreambuf::ofdnstreambuf()
:
    d_bufsize(0),
    d_buffer(0)
{}

inline ofdnstreambuf::ofdnstreambuf(int fd, size_t bufsize)
{
    open(fd, bufsize);
}
```

- The destructor calls the overridden function `sync()`, writing any characters stored in the output buffer to the device. If there's no buffer, the destructor needs to perform no actions:

```
inline ofdnstreambuf::~ofdnstreambuf()
{
    if (d_buffer)
    {
        sync();
        delete[] d_buffer;
    }
}
```

Even though the device is not closed in the above implementation this may not always be what one wants. It is left as an exercise to the reader to change this class in such a way that the device may optionally remain open. This approach was followed in, e.g., the `Bobcat library`[1]. See also section 20.1.2.2.

- The `open()` member initializes the buffer. Using `setp()`, the begin and end points of the buffer are set. This is used by the `streambuf` base class to initialize `pbase()`, `pptr()`, and `epptr()`:

```
inline void ofdnstreambuf::open(int fd, size_t bufsize)
```

[1] http://bobcat.sourceforge.net

```
{
    d_fd = fd;
    d_bufsize = bufsize == 0 ? 1 : bufsize;

    d_buffer = new char[d_bufsize];
    setp(d_buffer, d_buffer + d_bufsize);
}
```

- The member `sync()` will flush the as yet unflushed contents of the buffer to the device. Next, the buffer is reinitialized using `setp()`. Note that `sync()` returns 0 after a successful flush operation:

```
inline int ofdnstreambuf::sync()
{
    if (pptr() > pbase())
    {
        write(d_fd, d_buffer, pptr() - pbase());
        setp(d_buffer, d_buffer + d_bufsize);
    }
    return 0;
}
```

- Finally, the member `overflow()` is overridden. Since this member is called from the `streambuf` base class when the buffer is full, `sync()` is called first to flush the filled up buffer to the device. As this recreates an empty buffer, the character `c` which could not be written to the buffer by the `streambuf` base class 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 `streambuf` member functions, rather than doing it 'by hand', which might invalidate `streambuf`'s internal bookkeeping:

```
inline int ofdnstreambuf::overflow(int c)
{
    sync();
    if (c != EOF)
    {
        *pptr() = c;
        pbump(1);
    }
    return c;
}
```

- The member function implementations use low-level functions to operate on the file descriptors. So apart from `streambuf` the header file `unistd.h` must have been read by the compiler before the implementations of the member functions can be compiled.

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 <string>
#include <iostream>
#include <istream>
#include "fdout.h"
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;
    }
}
```

20.1.2 Classes for input operations

When classes to be used for input operation are derived from `std::streambuf`, they should be provided with an input buffer of at least one character. The one-character input buffer allows for the use of the member functions `istream::putback()` or `istream::ungetc()`. Stream classes (like `istream`) normally allow us to unget at least one character using their member functions `putback()` or `ungetc()`. This is important, as these stream classes usually interface to `streambuf` objects. Although it is strictly speaking not necessary to implement a buffer in classes derived from `streambuf`, using buffers in these cases is strongly advised: the implementation is very simple and straightforward, and the applicability of such classes will be greatly improved. Therefore, in all our classes derived from the class `streambuf` *at least* a buffer of one character will be defined.

20.1.2.1 Using a one-character buffer

When deriving a class (e.g., `ifdstreambuf`) from `streambuf` using a buffer of one character, at least its member `streambuf::underflow()` should be overridden, as this is the member to which all requests for input are eventually directed. Since a buffer is also needed, the member `streambuf::setg()` is used to inform the `streambuf` base class of the size of the input buffer, so that it is able to set up its input buffer pointers correctly. This will ensure that `eback()`, `gptr()`, and `egptr()` return correct values.

The required class shows the following characteristics:

- Like the class designed for output operations, this class is derived from `std::streambuf` as well. The class defines two data members, one of them a fixed-sized one character buffer. The data members are defined as protected data members so that derived classes (e.g., see section 20.1.2.3) can access them. Here is the full class interface:

```
class ifdstreambuf: public std::streambuf
{
    protected:
        int     d_fd;
        char    d_buffer[1];
    public:
        ifdstreambuf(int fd);
        int underflow();
};
```

- The constructor initializes the buffer. However, this initialization is done so that gptr() will be equal to egptr(). Since this implies that the buffer is empty, underflow() will immediately be called to refill the buffer:

```
inline ifdstreambuf::ifdstreambuf(int fd)
:
    d_fd(fd)
{
    setg(d_buffer, d_buffer + 1, d_buffer + 1);
}
```

- Finally underflow() is overridden. It will first ensure that the buffer is really empty. If not, then the next character in the buffer is returned. If the buffer is really empty, it is refilled by reading from the file descriptor. If this fails (for whatever reason), EOF is returned. More sophisticated implementations could react more intelligently here, of course. If the buffer could be refilled, setg() is called to set up streambuf's buffer pointers correctly.
- The implementations of the member functions use low-level functions to operate the file descriptors, so apart from streambuf the header file unistd.h must have been read by the compiler before the implementations of the member functions can be compiled.

This completes the construction of the ifdstreambuf class. It is used in the following program:

```
#include <iostream>
#include <istream>
#include <unistd.h>
#include "ifdbuf.h"
using namespace std;

int main(int argc)
{
    ifdstreambuf fds(STDIN_FILENO);
    istream      is(&fds);

    cout << is.rdbuf();
}
```

20.1.2.2 Using an n-character buffer

How complex would things get if we would decide to use a buffer of substantial size? Not that complex. The following class allows us to specify the size of a buffer, but apart from that it is basically the same class as `ifdstreambuf` developed in the previous section. To make things a bit more interesting, in the class `ifdnstreambuf` developed here, the member `streambuf::xsgetn()` is also overridden, to optimize reading of series of characters. Furthermore, a default constructor is provided which can be used in combination with the `open()` member to construct an `istream` object before the file descriptor becomes available. Then, once the descriptor becomes available, the `open()` member can be used to initiate the object's buffer. Later, in section 20.3, we'll encounter such a situation.

To save some space, the success of various calls was not checked. In 'real life' implementations, these checks should, of course, not be omitted. The class `ifdnstreambuf` has the following characteristics:

- Once again, it is derived from `std::streambuf`: Like the class `ifdstreambuf` (section 20.1.2.1), its data members are protected. Since the buffer's size is configurable, this size is kept in a dedicated data member, `d_bufsize`:

```
class ifdnstreambuf: public std::streambuf
{
    protected:
        int         d_fd;
        size_t    d_bufsize;
        char*       d_buffer;
    public:
        ifdnstreambuf();
        ifdnstreambuf(int fd, size_t bufsize = 1);
        ~ifdnstreambuf();
        void open(int fd, size_t bufsize = 1);
        int underflow();
        std::streamsize xsgetn(char *dest, std::streamsize n);
};
```

- The default constructor does not allocate a buffer, and can be used to construct an object before the file descriptor becomes known. A second constructor simply passes its arguments to `open()` which will then initialize the object so that it can actually be used:

```
inline ifdnstreambuf::ifdnstreambuf()
:
    d_bufsize(0),
    d_buffer(0)
{}
inline ifdnstreambuf::ifdnstreambuf(int fd, size_t bufsize)
{
    open(fd, bufsize);
}
```

- If the object has been initialized by `open()`, its destructor will both delete the object's buffer and use the file descriptor to close the device:

```
ifdnstreambuf::~ifdnstreambuf()
{
    if (d_bufsize)
    {
        close(d_fd);
        delete[] d_buffer;
    }
}
```

Even though the device is closed in the above implementation this may not always be what one wants. In cases where the open file descriptor is already available the intention may be to use that descriptor repeatedly, each time using a newly constructed `ifdnstreambuf` object. It is left as an exercise to the reader to change this class in such a way that the device may optionally be closed. This approach was followed in, e.g., the `Bobcat library`[2].

- The `open()` member simply allocates the object's buffer. It is assumed that the calling program has already opened the device. Once the buffer has been allocated, the base class member `setg()` is used to ensure that `eback()`, `gptr()`, and `egptr()` return correct values:

```
void ifdnstreambuf::open(int fd, size_t bufsize)
{
    d_fd = fd;
    d_bufsize = bufsize;
    d_buffer = new char[d_bufsize];
    setg(d_buffer, d_buffer + d_bufsize, d_buffer + d_bufsize);
}
```

- The overridden member `underflow()` is implemented almost identically to `ifdstreambuf`'s (section 20.1.2.1) member. The only difference is that the current class supports a buffer of larger sizes. Therefore, more characters (up to `d_bufsize`) may be read from the device at once:

```
int ifdnstreambuf::underflow()
{
    if (gptr() < egptr())
        return *gptr();

    int nread = read(d_fd, d_buffer, d_bufsize);

    if (nread <= 0)
        return EOF;

    setg(d_buffer, d_buffer, d_buffer + nread);
    return *gptr();
}
```

- Finally `xsgetn()` is overridden. In a loop, `n` is reduced until 0, at which point the function terminates. Alternatively, the member returns if `underflow()` fails to obtain more characters. This member optimizes the reading of series of characters: instead of calling `streambuf::sbumpc()` `n` times, a block of `avail` characters is copied to the destination, using `streambuf::gpumb()` to consume `avail` characters from the buffer using one function call:

```
std::streamsize ifdnstreambuf::xsgetn(char *dest, std::streamsize n)
{
    int nread = 0;

    while (n)
    {
        if (!in_avail())
        {
            if (underflow() == EOF)
                break;
```

[2] http://bobcat.sourceforge.net

```
            }

            int avail = in_avail();

            if (avail > n)
                avail = n;

            memcpy(dest + nread, gptr(), avail);
            gbump(avail);

            nread += avail;
            n -= avail;
        }

        return nread;
    }
```

- The implementations of the member functions use low-level functions to operate the file descriptors. So apart from `streambuf` the header file `unistd.h` must have been read by the compiler before the implementations of the member functions can be compiled.

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 <unistd.h>
#include <iostream>
#include <istream>
#include "ifdnbuf.h"
using namespace std;

int main(int argc)
{
                                    // internally: 30 char buffer
    ifdnstreambuf fds(STDIN_FILENO, 30);

    char buf[80];                   // main() reads blocks of 80
                                    // chars
    while (true)
    {
        size_t n = fds.sgetn(buf, 80);
        if (n == 0)
            break;
        cout.write(buf, n);
    }
}
```

20.1.2.3 Seeking positions in 'streambuf' objects

When devices support *seek operations*, classes derived from `streambuf` should override the members `streambuf::seekoff()` and `streambuf::seekpos()`. The class `ifdseek`, developed in this section, can be used to read information from devices supporting such seek operations. The class `ifdseek` was derived from `ifdstreambuf`, so it uses a character buffer of just one character. The facilities to perform seek operations, which are added to our new class `ifdseek`, 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 in such a way that its second and third parameters point beyond the available input buffer. Let's have a look at the characteristics of `ifdseek`:

- As mentioned, `ifdseek` is derived from `ifdstreambuf`. Like the latter class, `ifdseek`'s member functions use facilities declared in `unistd.h`. So, the compiler must have seen `unistd.h` before it can compile the class's members functions. To reduce the amount of typing when specifying types and constants from `std::streambuf` and `std::ios`, several `typedefs` are defined at the class's very top. These typedefs refer to types that are defined in the header file `ios`, which must therefore be included as well before the compiler reads `ifdseek`'s class definition. Here is the class's interface:

```
class ifdseek: public ifdstreambuf
{
    typedef std::streambuf::pos_type        pos_type;
    typedef std::streambuf::off_type        off_type;
    typedef std::ios::seekdir               seekdir;
    typedef std::ios::openmode              openmode;

    public:
        ifdseek(int fd);
        pos_type seekoff(off_type offset, seekdir dir, openmode);
        pos_type seekpos(pos_type offset, openmode mode);
};
```

- The class is given a rather basic implementation. The only required constructor expects the device's file descriptor. It has no special tasks to perform and only needs to call its base class constructor:

```
inline ifdseek::ifdseek(int fd)
:
    ifdstreambuf(fd)
{}
```

- The member `seek_off()` is responsible for performing the actual seek operations. It calls `lseek()` to seek a new position in a device whose file descriptor is known. If seeking succeeds, `setg()` is called to define an already empty buffer, so that the base class's `underflow()` member will refill the buffer at the next input request.

```
ifdseek::pos_type ifdseek::seekoff(off_type off, seekdir dir, openmode)
{
    pos_type pos =
        lseek
        (
            d_fd, off,
            (dir ==  std::ios::beg) ? SEEK_SET :
            (dir ==  std::ios::cur) ? SEEK_CUR :
                                      SEEK_END
        );

    if (pos < 0)
        return -1;
```

```
        setg(d_buffer, d_buffer + 1, d_buffer + 1);
        return pos;
    }
```

- Finally, the companion function `seekpos` is overridden as well: it is actually defined as a call to `seekoff()`:

```
    inline ifdseek::pos_type ifdseek::seekpos(pos_type off, openmode mode)
    {
        return seekoff(off, std::ios::beg, mode);
    }
```

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 with the exception of the first line, every other line is shown twice:

```
#include "fdinseek.h"
#include <string>
#include <iostream>
#include <istream>
#include <iomanip>
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;
        }
    }
}
```

20.1.2.4 Multiple 'unget()' calls in 'streambuf' objects

As mentioned before, `streambuf` classes and classes derived from `streambuf` should support *at least* ungetting the last read character. Special care must be taken when *series* of `unget()` calls must be supported. In this section the construction of a class supporting a configurable number of `istream::unget()` or `istream::putback()` calls is discussed.

Support for multiple (say 'n') `unget()` calls is realized by reserving an initial section of the input buffer, which is gradually filled up to contain the last `n` characters read. The class was implemented as follows:

- Once again, the class is derived from `std::streambuf`. It defines several data members, allowing the class to perform the bookkeeping required to maintain an unget-buffer of a configurable size:

```
class fdunget: public std::streambuf
{
    int         d_fd;
    size_t    d_bufsize;
    size_t    d_reserved;
    char*       d_buffer;
    char*       d_base;

    public:
        fdunget(int fd, size_t bufsz, size_t unget);
        ~fdunget();
        int underflow();
};
```

- The class's constructor expects a file descriptor, a buffer size and the number of characters that can be ungot or pushed back as its arguments. This number determines the size of a *reserved* area, defined as the first `d_reserved` bytes of the class's input buffer.
  - The input buffer will always be at least one byte larger than `d_reserved`. So, a certain number of bytes may be read. Then, once `reserved` bytes have been read at least `reserved` bytes can be ungot.
  - Next, the starting point for reading operations is configured: it is called `d_base`, pointing to a location `reserved` bytes from the start of `d_buffer`. This will always be the point where the buffer refills start.
  - Now that the buffer has been constructed, we're ready to define `streambuf`'s buffer pointers using `setg()`. As no characters have been read yet, all pointers are set to point to `d_base`. If `unget()` is called at this point, no characters are available, so `unget()` will (correctly) fail.
  - Eventually, the refill buffer's size is determined as the number of allocated bytes minus the size of the reserved area.

  Here is the class's constructor:

```
fdunget::fdunget(int fd, size_t bufsz, size_t unget)
:
    d_fd(fd),
    d_reserved(unget)
{
    size_t allocate =
```

```
                bufsz > d_reserved ?
                    bufsz
                :
                    d_reserved + 1;

        d_buffer = new char[allocate];

        d_base = d_buffer + d_reserved;
        setg(d_base, d_base, d_base);

        d_bufsize = allocate - d_reserved;
    }
```

- The class's destructor simply returns the memory allocated for the buffer to the common pool:

```
    inline fdunget::~fdunget()
    {
        delete[] d_buffer;
    }
```

- Finally, `underflow()` is overridden.
  - Firstly, the standard check to determine whether the buffer is really empty is applied.
  - If empty, it determines the number of characters that could potentially be ungot. At this point, the input buffer is exhausted. So this value may be any value between 0 (the initial state) or the input buffer's size (when the reserved area has been filled up completely, and all current characters in the remaining section of the buffer have also been read).
  - Next the number of bytes to move into the reserved area is computed. This number is at most `d_reserved`, but it is equal to the actual number of characters that can be ungot if this value is smaller.
  - Now that the number of characters to move into the reserved area is known, this number of characters is moved from the input buffer's end to the area immediately before `d_base`.
  - Then the buffer is refilled. This all is standard, but notice that reading starts from `d_base` and not from `d_buffer`.
  - Finally, `streambuf`'s read buffer pointers are set up. `Eback()` is set to `move` locations before `d_base`, thus defining the guaranteed unget-area, `gptr()` is set to `d_base`, since that's the location of the first read character after a refill, and `egptr()` is set just beyond the location of the last character read into the buffer.

Here is `underflow()`'s implementation:

```
    int fdunget::underflow()
    {
        if (gptr() < egptr())
            return *gptr();

        size_t ungetsize = gptr() - eback();
        size_t move = std::min(ungetsize, d_reserved);

        memcpy(d_base - move, egptr() - move, move);

        int nread = read(d_fd, d_base, d_bufsize);
        if (nread <= 0)       // none read -> return EOF
            return EOF;
```

```
        setg(d_base - move, d_base, d_base + nread);

        return *gptr();
    }
```

The following program illustrates the `class fdunget`. It reads at most 10 characters from the standard input, stopping at `EOF`. A guaranteed unget-buffer of 2 characters is defined in a buffer holding 3 characters. Just before reading a character, the program tries to unget at most 6 characters. This is, of course, not possible; but the program will nicely 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
*/
```

20.2 Fixed-sized field extraction from istream objects

Usually when extracting information from `istream` objects `operator>>()`, the standard extraction operator, is perfectly suited for the task as in most cases the extracted fields are white-space or otherwise clearly separated from each other. But this does not hold true in all situations. For example, when a web-form is posted to some processing script or program, the receiving program may receive the form field's values as *url-encoded* characters: letters and digits are sent unaltered, blanks are sent as `+` characters, and all other characters start with `%` followed by the character's ascii-value represented by its two digit hexadecimal value.

When decoding url-encoded information, a simple hexadecimal extraction won't work, since that will extract as many hexadecimal characters as available, instead of just two. Since the letters `a-f` and `0-9` are legal hexadecimal characters, a text like `My name is ‘Ed’`, url-encoded as

```
My+name+is+%60Ed%27
```

will result in the extraction of the hexadecimal values `60ed` and `27`, instead of `60` and `27`. The name

Ed will disappear from view, which is clearly not what we want.

In this case, having seen the %, we could extract 2 characters, put them in an istringstream object, and extract the hexadecimal value from the istringstream object. A bit cumbersome, but doable. Other approaches, however, are possible as well.

The following class fistream for *fixed-sized field istream* defines an istream class supporting both fixed-sized field extractions and blank-delimited extractions (as well as unformatted read() calls). The class may be initialized as a *wrapper* around an existing istream, or it can be initialized using the name of an existing file. The class is derived from istream, allowing all extractions and operations supported by istreams in general. The class will need the following data members:

- d_filebuf: a filebuffer used when fistream reads its information from a named (existing) file. Since the filebuffer is only needed in that case, and since it must be allocated dynamically, it is defined as an auto_ptr<filebuf> object.
- d_streambuf: a pointer to fistream's streambuf. It will point to filebuf when fistream opens a file by name. When an existing istream is used to construct an fistream, it will point to the existing istream's streambuf.
- d_iss: an istringstream object which is used for the fixed field extractions.
- d_width: an size_t indicating the width of the field to extract. If 0 no fixed field extractions will be used, but information will be extracted from the istream base class object using standard extractions.

Here is the initial section of fistream's class interface:

```
    class fistream: public std::istream
    {
        std::auto_ptr<std::filebuf> d_filebuf;
        std::streambuf *d_streambuf;
        std::istringstream d_iss;
        size_t d_width;
```

As mentioned, fistream objects can be constructed from either a filename or an existing istream object. Thus, the class interface shows two constructors:

```
            fistream(std::istream &stream);
            fistream(char const *name,
                std::ios::openmode mode = std::ios::in);
```

When an fistream object is constructed using an existing istream object, the fistream's istream part will simply use the stream's streambuf object:

```
fistream::fistream(istream &stream)
:
    istream(stream.rdbuf()),
    d_streambuf(rdbuf()),
    d_width(0)
{}
```

When an fstream object is constructed using a filename, the istream base initializer is given a new filebuf object to be used as its streambuf. Since the class's data members are not initialized before the class's base class has been constructed, `d_filebuf` can only be initialized thereafter. By then, the `filebuf` is only available as `rdbuf()`, which returns a `streambuf`. However, as it is actually a `filebuf`, a `reinterpret_cast` is used to cast the `streambuf` pointer returned by `rdbuf()` to a `filebuf *`, so `d_filebuf` can be initialized:

```
fistream::fistream(char const *name, ios::openmode mode)
:
    istream(new filebuf()),
    d_filebuf(reinterpret_cast<filebuf *>(rdbuf())),
    d_streambuf(d_filebuf.get()),
    d_width(0)
{
    d_filebuf->open(name, mode);
}
```

There is only one additional public member: `setField(field const &)`. This member is used to define the size of the next field to extract. Its parameter is a reference to a `field` class, a *manipulator class* defining the width of the next field.

Since a `field &` is mentioned in `fistream`'s interface, `field` must be declared before `fistream`'s interface starts. The class `field` itself is simple: it declares `fistream` as its friend, and it has two data members: `d_width` specifies the width of the next field, `d_newWidth` is set to `true` if `d_width`'s value should actually be used. If `d_newWidth` is false, `fistream` will return to its standard extraction mode. The class `field` furthermore has two constructors: a default constructor, setting `d_newWidth` to `false` and a second constructor expecting the width of the next field to extract as its value. Here is the class `field`:

```
    class field
    {
        friend class fistream;
        size_t d_width;
        bool     d_newWidth;

        public:
            field(size_t width);
            field();
    };

    inline field::field(size_t width)
    :
        d_width(width),
        d_newWidth(true)
    {}

    inline field::field()
    :
        d_newWidth(false)
    {}
```

Since `field` declares `fistream` as its friend, `setField` may inspect `field`'s members directly.

Time to return to `setField()`. This function expects a reference to a `field` object, initialized in either of three different ways:

- `field()`: When `setField()`'s argument is a `field` object constructed by its default constructor the next extraction will use the same fieldwidth as the previous extraction.
- `field(0)`: When this `field` object is used as `setField()`'s argument, fixed-sized field extraction stops, and the `fistream` will act like any standard `istream` object.
- `field(x)`: When the `field` object itself is initialized by a non-zero size_t value `x`, then the next field width will be `x` characters wide. The preparation of such a field is left to `setBuffer()`, `fistream`'s only private member.

Here is `setField()`'s implementation:

```
std::istream &fistream::setField(field const &params)
{
    if (params.d_newWidth)                  // new field size requested
        d_width = params.d_width;           // set new width

    if (!d_width)                           // no width?
        rdbuf(d_streambuf);                 // return to the old buffer
    else
        setBuffer();                        // define the extraction buffer

    return *this;
}
```

The private member `setBuffer()` defines a buffer of `d_width + 1` characters, and uses `read()` to fill the buffer with `d_width` characters. The buffer is terminated by an `ASCII-Z` character. This buffer is then used to initialize the `d_str` member. Finally, `fistream`'s `rdbuf()` member is used to extract the `d_str`'s data via the `fistream` object itself:

```
void fistream::setBuffer()
{
    char *buffer = new char[d_width + 1];

    rdbuf(d_streambuf);                     // use istream's buffer to
    buffer[read(buffer, d_width).gcount()] = 0; // read d_width chars,
                                                // terminated by ascii-Z
    d_iss.str(buffer);
    delete buffer;

    rdbuf(d_iss.rdbuf());                   // switch buffers
}
```

Although `setField()` could be used to configure `fistream` to use or not to use fixed-sized field extraction, using manipulators is probably preferable. To allow `field` objects to be used as manipulators, an overloaded extraction operator was defined, accepting an `istream &` and a `field const &` object. Using this extraction operator, statements like

```
fis >> field(2) >> x >> field(0);
```

are possible (assuming `fis` is a `fistream` object). Here is the overloaded `operator>>()`, as well as its declaration:

```
istream &std::operator>>(istream &str, field const &params)
```

```
{
    return reinterpret_cast<fistream *>(&str)->setField(params);
}
```

Declaration:

```
namespace std
{
    istream &operator>>(istream &str, FBB::field const &params);
}
```

Finally, an example. The following program uses a `fistream` object to url-decode url-encoded information appearing at its standard input:

```
    int main()
    {
        fistream fis(cin);

        fis >> hex;
        while (true)
        {
            size_t x;
            switch (x = fis.get())
            {
                case '\n':
                    cout << endl;
                break;
                case '+':
                    cout << ' ';
                break;
                case '%':
                    fis >> field(2) >> x >> field(0);
                // FALLING THROUGH
                default:
                    cout << static_cast<char>(x);
                break;
                case EOF:
                return 0;
            }
        }
    }
    /*
        Generated output after:
            echo My+name+is+%60Ed%27 | a.out

        My name is ‘Ed’
    */
```

20.3 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()`: the return value in the original process (called the *parent process*) differs from the return value in the newly created process (called the *child process*):

- In the *parent process* `fork()` returns the *process ID* of the child process 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 all bookkeeping details of a system call like `fork()` from its users. The class `Fork` developed here 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 will do. However, `Fork` objects should be able to activate their child processes.

Unfortunately, `Fork`'s constructor cannot know what actions its child process should perform. Similarly, it cannot know what actions the parent process should perform. For this particular situation, the *template method design pattern* was developed. According to Gamma c.s., the *template method design pattern*

> "Define(s) the skeleton of an algorithm in an operation, deferring some steps to subclasses. (The) Template Method (design pattern) lets subclasses redefine certain steps of an algorithm, without changing the algorithm's structure."

This design pattern allows us to define an *abstract base class* already implementing the essential steps related to the `fork()` system call and deferring the implementation of certain normally used parts of the `fork()` system call to subclasses.

The `Fork` abstract base class itself has the following characteristics:

- It defines a data member `d_pid`. This data member will contain the child's *process id* (in the parent process) and the value 0 in the child process. Its public interface declares but two members:
  - a `fork()` member function, performing the actual forking (i.e., it will create the (new) child process);
  - an *empty* `virtual` destructor `~Fork()`, which will be overridden by derived classes defining their own destructors.

    ```
    inline Fork::~Fork()
    {}
    ```

Here is `Fork`'s interface:

```
class Fork
{
    int d_pid;

    public:
        virtual ~Fork();
        void fork();

    protected:
        int pid() const;
        virtual void childRedirections();
        virtual void parentRedirections();
        virtual void childProcess() = 0;    // both must be implemented
        virtual void parentProcess() = 0;
        int waitForChild();                 // returns the status
};
```

- All remaining member functions are declared in the class's `protected` section and can thus *only* be used by derived classes. They are:
  - The member function `pid()`, allowing derived classes to access the system `fork()`'s return value:

    ```
    inline int Fork::pid() const
    {
        return d_pid;
    }
    ```

  - A member `int waitForChild()`, which can be called by parent processes to wait for the completion of their child processes (as discussed below). This member is declared in the class interface. Its implementation is

    ```
    #include "fork.ih"

    int Fork::waitForChild()
    {
        int status;

        waitpid(d_pid, &status, 0);

        return WEXITSTATUS(status);
    }
    ```

    This simple implementation returns the child's *exit status* to the parent. The called system function `waitpid()` *blocks* until the child terminates.
  - When `fork()` system calls are used, *parent processes* and *child processes* may always be distinguished. The main distinction between these processes is that `d_pid` will be equal to the child's process-id in the parent process, while `d_pid` will be equal to 0 in the child process itself. Since these two processes may always be distinguished, they must be implemented by classes derived from `Fork`. To enforce this requirement, the members `childProcess()`, defining the child process' actions and `parentProcess()`, defining the parent process' actions we defined as pure virtual functions:
  - In addition, communication between parent- and child processes may use standard streams or other facilities, like *pipes* (cf. section 20.3.3). To facilitate this inter-process communication, derived classes *may* implement:

  * `childRedirections()`: this member should be implemented if any standard stream (`cin, cout`) or `cerr` must be redirected in the *child* process (cf. section 20.3.1);
  * `parentRedirections()`: this member should be implemented if any standard stream (`cin, cout`) or `cerr` must be redirected in the *parent* process.

  Redirection of the standard streams will be necessary if parent- and child processes should communicate with each other via the standard streams. Here are their default definitions provided by the class's interface:

```
inline void Fork::childRedirections()
{}
inline void Fork::parentRedirections()
{}
```

The member function `fork()` calls the system function `fork()` (Caution: since the system function `fork()` is called by a member function having the same name, the `::` scope resolution operator must be used to prevent a recursive call of the member function itself). After calling `::fork()`, depending on its return value, either `parentProcess()` or `childProcess()` is called. Maybe redirection is necessary. `Fork::fork()`'s implementation calls `childRedirections()` just before calling `childProcess()`, and `parentRedirections()` just before calling `parentProcess()`:

```
#include "fork.ih"

void Fork::fork()
{
    if ((d_pid = ::fork()) < 0)
        throw "Fork::fork() failed";

    if (d_pid == 0)                 // childprocess has pid == 0
    {
        childRedirections();
        childProcess();

        exit(1);                    // we shouldn't come here:
                                    // childProcess() should exit
    }

    parentRedirections();
    parentProcess();
}
```

In `fork.cc` the class's *internal header file* `fork.ih` is included. This header file takes care of the inclusion of the necessary system header files, as well as the inclusion of `fork.h` itself. Its implementation is:

```
#include "fork.h"
#include <cstdlib>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

Child processes should not return: once they have completed their tasks, they should terminate. This happens automatically when the child process performs a call to a member of the `exec...()` family, but if the child itself remains active, then it must make sure that it terminates properly.

A child process normally uses `exit()` to terminate itself, but it should be realized that `exit()` prevents the activation of destructors of objects defined at the same or more superficial nesting levels than the level at which `exit()` is called. Destructors of globally defined objects *are* activated when `exit()` is used. When using `exit()` to terminate `childProcess()`, it should either itself call a support member function defining all nested objects it needs, or it should define all its objects in a compound statement (e.g., using a `throw` block) calling `exit()` beyond the compound statement.

Parent processes should normally wait for their children to complete. The terminating child processes inform their parent that they are about to terminate by sending out a *signal* which should be caught by their parents. If child processes terminate and their parent processes do not catch those signal then such child processes remain visible as so-called *zombie* processes.

If parent processes must wait for their children to complete, they may call the member `waitForChild()`. This member returns the exit status of a child process to its parent.

There exists a situation where the *child* process *continues* to live, but the *parent* dies. In nature this happens all the time: parents tend to die before their children do. In our context (i.e. **C++**), this is called a *daemon* program: the parent process dies and the child program continues to run as a child of the basic `init` process. Again, when the child eventually dies a signal is sent to its 'step-parent' `init`. No zombie is created here, as `init` catches the termination signals of all its (step-) children. The construction of a daemon process is very simple, given the availability of the class `Fork` (cf. section 20.3.2).

20.3.1 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 terminates, 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 program at the end of this section 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.

Actually, this doesn't happen; and `hello world` still appears at the program's standard output instead of the redirected file. A solution of this problem involves redirection at the operating system level, for which some operating systems (e.g., Unix and friends) provide system calls like `dup()` and `dup2()`. Examples of these system calls are given in section 20.3.3.

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
*/
```

20.3.2 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 requires no features other than those currently offered by the class `Fork`. Here is an example illustrating the construction of a daemon program:

```
#include <iostream>
#include <unistd.h>
#include "fork.h"

class Daemon: public Fork
{
    public:
        virtual void parentProcess()        // the parent does nothing.
        {}

        virtual void childProcess()
        {
            sleep(3);                       // actions taken by the child
                                            // just a message...
            std::cout << "Hello from the child process\n";
            exit (0);                       // The child process exits.
        }
};
```

```
int main()
{
    Daemon daemon;

    daemon.fork();          // program immediately returns
    return 0;
}

/*
    Generated output:
The next command prompt, then after 3 seconds:
Hello from the child process
*/
```

20.3.3 The class ‘Pipe’

Redirection at the system level involves the use of *file descriptors*, created by the `pipe()` system call. When two processes want to communicate using such file descriptors, the following takes place:

- The process constructs two *associated file descriptors* using the `pipe()` system call. One of the file descriptors is used for writing, the other file descriptor is used for reading.
- Forking takes place (i.e., the system `fork()` function is called), duplicating the file descriptors. Now we have four file descriptors as both the child process and the parent process have their own copies of the two file descriptors created by `pipe()`.
- One process (say, the parent process) will use the filedescriptors for *reading*. It should close its filedescriptor intended for *writing*.
- The other process (say, the child process) will use the filedescriptors for *writing*. It should close its filedescriptor intended for *reading*.
- All information written by the child process to the file descriptor intended for writing, can now be read by the parent process from the corresponding file descriptor intended for reading, thus establishing a communication channel between the child- and the parent process.

Though basically simple, errors may easily creep in: purposes of file descriptors available to the two processes (child- or parent-) may easily get mixed up. To prevent bookkeeping errors, the bookkeeping may be properly set up once, to be hidden therafter inside a class like the `Pipe` class constructed here. Let’s have a look at its characteristics (before the implementations can be compiled, the compiler must have read the class’s header file as well as the file `unistd.h`):

- The `pipe()` system call expects a pointer to two `int` values, which will represent, respectively, the file descriptors to use for accessing the *reading end* and the *writing end* of the constructed pipe, after `pipe()`’s successful completion. To avoid confusion, an `enum` is defined associating these ends with symbolic constants. Furthermore, the class stores the two file descriptors in a data member `d_fd`. Here is the class header and its private data:

  ```
  class Pipe
  {
      enum    RW { READ, WRITE };
      int     d_fd[2];
  ```

- The class only needs a default constructor. This constructor calls `pipe()` to create a set of associated file descriptors used for accessing both ends of a pipe:

```
Pipe::Pipe()
{
    if (pipe(d_fd))
        throw "Pipe::Pipe(): pipe() failed";
}
```

- The members `readOnly()` and `readFrom()` are used to configure the pipe's reading end. The latter function is used to set up redirection, by providing an alternate file descriptor which can be used to read from the pipe. Usually this alternate file descriptor is `STDIN_FILENO`, allowing `cin` to extract information from the pipe. The former function is merely used to configure the reading end of the pipe: it closes the matching writing end, and returns a file descriptor that can be used to read from the pipe:

```
int Pipe::readOnly()
{
    close(d_fd[WRITE]);
    return d_fd[READ];
}
void Pipe::readFrom(int fd)
{
    readOnly();

    redirect(d_fd[READ], fd);
    close(d_fd[READ]);
}
```

- `writeOnly()` and two `writtenBy()` members are available to configure the writing end of a pipe. The former function is merely used to configure the writing end of the pipe: it closes the matching reading end, and returns a file descriptor that can be used to write to the pipe:

```
int Pipe::writeOnly()
{
    close(d_fd[READ]);
    return d_fd[WRITE];
}
void Pipe::writtenBy(int fd)
{
    writtenBy(&fd, 1);
}
void Pipe::writtenBy(int const *fd, size_t n)
{
    writeOnly();

    for (size_t idx = 0; idx < n; idx++)
        redirect(d_fd[WRITE], fd[idx]);

    close(d_fd[WRITE]);
}
```

  For the latter member two overloaded versions are available:

  - `writtenBy(int fileDescriptor)` is used to configure *single* redirection, so that a specific file descriptor (usually `STDOUT_FILENO` or `STDERR_FILENO`) may be used to write to the pipe;

  - (`writtenBy(int *fileDescriptor, size_t n = 2)`) may be used to configure *multiple* redirection, providing an array argument containing file descriptors. Information written to any of these file descriptors is actually written into the pipe.

- The class has one private data member, `redirect()`, which is used to define a redirection using the `dup2()` system call. This function expects two file descriptors. The first file descriptor represents a file descriptor which can be used to access the device's information, the second file descriptor is an alternate file descriptor which may also be used to access the device's information once `dup2()` has completed successfully. Here is `redirect()`'s implementation:

```
void Pipe::redirect(int d_fd, int alternateFd)
{
    if (dup2(d_fd, alternateFd) < 0)
        throw "Pipe: redirection failed";
}
```

Now that redirection can be configured easily using one or more `Pipe` objects, we'll now use `Fork` and `Pipe` in several demonstration programs.

20.3.4 The class 'ParentSlurp'

The class `ParentSlurp`, derived from `Fork`, starts a child process which *execs* a program (like `/bin/ls`). The (standard) output of the execed program is then read by the parent process. The parent process will (for demonstration purposes) write the lines it receives to its standard output stream, while prepending linenumbers to the received lines. It is most convenient here to redirect the parents standard input stream, so that the parent can read the *output* from the child process from its `std::cin` *input* stream. Therefore, the only pipe that's used is used as an *input* pipe at the parent, and an *output* pipe at the child.

The class `ParentSlurp` has the following characteristics:

- It is derived from `Fork`. Before starting `ParentSlurp`'s class interface, the compiler must have read both `fork.h` and `pipe.h`. Furthermore, the class only uses one data member: a `Pipe` object `d_pipe`.
- Since `Pipe`'s constructor automatically constructs a pipe, and since `d_pipe` is automatically constructed by `ParentSlurp`'s default constructor, there is no need to define `ParentSlurp`'s constructor explicitly. As no construtor needs to be implemented, all `ParentSlurp`'s members can be declared as `protected` members. Here is the class's interface:

```
class ParentSlurp: public Fork
{
    Pipe    d_pipe;

    protected:
        virtual void childRedirections();
        virtual void parentRedirections();
        virtual void childProcess();
        virtual void parentProcess();
};
```

- The `childRedirections()` member configures the pipe as a pipe for reading. So, all information written to the child's standard output stream will end up in the pipe. The big advantage of this all is that no streams around file descriptors are needed to write to a file descriptor:

```
inline void ParentSlurp::childRedirections()
{
    d_pipe.writtenBy(STDOUT_FILENO);
}
```

- The `parentRedirections()` member, configures its end of the pipe as a reading pipe. It does so by redirecting the reading end of the pipe to its standard input file descriptor (`STDIN_FILENO`), thus allowing extractions from `cin` instead of using streams built around file descriptors.

```
inline void ParentSlurp::parentRedirections()
{
    d_pipe.readFrom(STDIN_FILENO);
}
```

- The `childProcess()` member only has to concentrate on its own actions. As it only needs to execute a program (writing information to its standard output), the member consists of but one statement:

```
inline void ParentSlurp::childProcess()
{
    execl("/bin/ls", "/bin/ls", static_cast<char *>(0));
}
```

- The `parentProcess()` member simply 'slurps' the information appearing at its standard input. Doing so, it actually reads the child's output. It copies the received lines to its standard output stream after having prefixed line numbers to them:

```
void ParentSlurp::parentProcess()
{
    std::string     line;
    size_t    nr = 1;

    while (getline(std::cin, line))
        std::cout << nr++ << ": " << line << std::endl;

    waitForChild();
}
```

The following program simply constructs a `ParentSlurp` object, and calls its `fork()` member. Its output consists of a numbered list of files in the directory where the program is started. Note that the program also needs the `fork.o, pipe.o` and `waitforchild.o` object files (see earlier sources):

```
int main()
{
    ParentSlurp ps;

    ps.fork();
    return 0;
}
/*
    Generated Output (example only, actually obtained output may differ):

    1: a.out
    2: bitand.h
```

```
        3: bitfunctional
        4: bitnot.h
        5: daemon.cc
        6: fdinseek.cc
        7: fdinseek.h
        ...
    */
```

20.3.5 Communicating with multiple children

The next step up the ladder is the construction of a child-process monitor. Here, the parent process is responsible for all its child processes, but it also must read their standard output. The user may enter information at the parent process' standard input, for which a simple *command language* is defined:

- start will start a new child process. The parent will return the ID (a number) to the user. The ID may thereupon be used to send a message to that particular child process
- <nr> text will send "text" to the child process having ID <nr>;
- stop <nr> will terminate the child process having ID <nr>;
- exit will terminate the parent as well as all of its children.

Furthermore, the child process that hasn't received text for some time will complain, by sending a message to the parent-process. The parent process will then simply transmit the received message to the user, by copying it to the standard output stream.

A problem with programs like our monitor is that these programs allow *asynchronous input* from multiple sources: input may appear at the standard input as well as at the input-sides of pipes. Also, multiple output channels are used. To handle situations like these, the select() system call was developed.

20.3.5.1 The class 'Select'

The select() system call was developed to handle asynchronous *I/O multiplexing*. This system call can be used to handle, e.g., input appearing simultaneously at a set of file descriptors.

The select() system function is rather complex, and its full discussion is beyond the **C++** Annotations' scope. However, its use may be simplified by providing a class Selector, hiding its details and offering an easy-to-use public interface. Here its characteristics are discussed:

- Most of Select's members are very small, allowing us to define most of its members as inline functions. The class requires quite a few data members. Most of them of types that were specifically constructed for use by select(). Therefore, before the class interface can be handled by the compiler, various header files must have been read by it:

```
#include <limits.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
```

- The class definition and its data members may appear next. The data type `fd_set` is a type designed to be used by `select()` and variables of this type contain the set of filedescriptors on which `select()` has sensed some activity. Furthermore, `select()` allows us to fire an *asynchronous alarm*. To specify alarm times, the class receives a `timeval` data member. The remaining members are used by the class for internal bookkeeping purposes, illustrated below. Here is the class's interface:

```
class Selector
{
    fd_set          d_read;
    fd_set          d_write;
    fd_set          d_except;
    fd_set          d_ret_read;
    fd_set          d_ret_write;
    fd_set          d_ret_except;
    timeval         d_alarm;
    int             d_max;
    int             d_ret;
    int             d_readidx;
    int             d_writeidx;
    int             d_exceptidx;

    public:
        Selector();

        int wait();
        int nReady();
        int readFd();
        int writeFd();
        int exceptFd();
        void setAlarm(int sec, int usec = 0);
        void noAlarm();
        void addReadFd(int fd);
        void addWriteFd(int fd);
        void addExceptFd(int fd);
        void rmReadFd(int fd);
        void rmWriteFd(int fd);
        void rmExceptFd(int fd);

    private:
        int checkSet(int *index, fd_set &set);
        void addFd(fd_set *set, int fd);
};
```

The following member functions are part of the class's public interface:

- `Selector()`: the (default) constructor. It clears the read, write, and execute `fd_set` variables, and switches off the alarm. Except for `d_max`, the remaining data members do not require initializations. Here is the implementation of `Selector`'s constructor:

```
Selector::Selector()
{
    FD_ZERO(&d_read);
    FD_ZERO(&d_write);
```

```
        FD_ZERO(&d_except);
        noAlarm();
        d_max = 0;
    }
```

- `int wait()`: this member function will *block()* until activity is sensed at any of the file descriptors monitored by the `Selector` object, or if the *alarm* times out. It will throw an exception when the `select()` system call itself fails. Here is `wait()`'s implementation:

```
    int Selector::wait()
    {
        timeval t = d_alarm;

        d_ret_read = d_read;
        d_ret_write = d_write;
        d_ret_except = d_except;

        d_readidx = 0;
        d_writeidx = 0;
        d_exceptidx = 0;

        d_ret = select(d_max, &d_ret_read, &d_ret_write, &d_ret_except, &t);

        if (d_ret < 0)
            throw "Selector::wait()/select() failed";

        return d_ret;
    }
```

- `int nReady`: this member function's return value is defined only when `wait()` has returned. In that case it returns 0 for a alarm-timeout, -1 if `select()` failed, and the number of file descriptors on which activity was sensed otherwise. It can be implemented inline:

```
    inline int Selector::nReady()
    {
        return d_ret;
    }
```

- `int readFd()`: this member function's return value also is defined only after `wait()` has returned. Its return value is -1 if no (more) input file descriptors are available. Otherwise the next file descriptor available for reading is returned. Its inline implementation is:

```
    inline int Selector::readFd()
    {
        return checkSet(&d_readidx, d_ret_read);
    }
```

- `int writeFd()`: operating analogously to `readFd()`, it returns the next file descriptor to which output is written. Using `d_writeidx` and `d_ret_read`, it is implemented analogously to `readFd()`;

- `int exceptFd()`: operating analogously to `readFd()`, it returns the next exception file descriptor on which activity was sensed. Using `d_except_idx` and `d_ret_except`, it is implemented analogously to `readFd()`;

- `void setAlarm(int sec, int usec = 0)`: this member activates `Select`'s alarm facility. At least the number of seconds to wait for the alarm to go off must be specified. It simply assigns values to `d_alarm`'s fields. Then, at the next `Select::wait()` call, the alarm will fire (i.e., `wait()` returns with return value 0) once the configured alarm-interval has passed. Here is its (inline) implementation:

```
inline void Selector::setAlarm(int sec, int usec)
{
    d_alarm.tv_sec  = sec;
    d_alarm.tv_usec = usec;
}
```

- `void noAlarm()`: this member switches off the alarm, by simply setting the alarm interval to a very long period. Implemented inline as:

```
inline void Selector::noAlarm()
{
    setAlarm(INT_MAX, INT_MAX);
}
```

- `void addReadFd(int fd)`: this member adds a file descriptor to the set of input file descriptors monitored by the `Selector` object. The member function `wait()` will return once input is available at the indicated file descriptor. Here is its inline implementation:

```
inline void Selector::addReadFd(int fd)
{
    addFd(&d_read, fd);
}
```

- `void addWriteFd(int fd)`: this member adds a file descriptor to the set of output file descriptors monitored by the `Selector` object. The member function `wait()` will return once output is available at the indicated file descriptor. Using `d_write`, it is implemented analogously as `addReadFd()`;

- `void addExceptFd(int fd)`: this member adds a file descriptor to the set of exception file descriptors to be monitored by the `Selector` object. The member function `wait()` will return once activity is sensed at the indicated file descriptor. Using `d_except`, it is implemented analogously as `addReadFd()`;

- `void rmReadFd(int fd)`: this member removes a file descriptor from the set of input file descriptors monitored by the `Selector` object. Here is its inline implementation:

```
inline void Selector::rmReadFd(int fd)
{
    FD_CLR(fd, &d_read);
}
```

- `void rmWriteFd(int fd)`: this member removes a file descriptor from the set of output file descriptors monitored by the `Selector` object. Using `d_write`, it is implemented analogously as `rmReadFd()`;

- `void rmExceptFd(int fd)`: this member removes a file descriptor from the set of exception file descriptors to be monitored by the `Selector` object. Using `d_except`, it is implemented analogously as `rmReadFd()`;

The class's remaining (two) members are support members, and should not be used by non-member functions. Therefore, they should be declared in the class's `private` section:

- The member `addFd()` adds a certain file descriptor to a certain `fd_set`. Here is its implementation:

```
void Selector::addFd(fd_set *set, int fd)
{
    FD_SET(fd, set);
    if (fd >= d_max)
        d_max = fd + 1;
}
```

- The member `checkSet()` tests whether a certain file descriptor (`*index`) is found in a certain `fd_set`. Here is its implementation:

```
int Selector::checkSet(int *index, fd_set &set)
{
    int &idx = *index;

    while (idx < d_max && !FD_ISSET(idx, &set))
        ++idx;

    return idx == d_max ? -1 : idx++;
}
```

20.3.5.2 The class 'Monitor'

The `monitor` program uses a `Monitor` object to do most of the work. The class has only one public constructor and one public member, `run()`, to perform its tasks. Therefore, all other member functions described below should be declared in the class's `private` section.

`Monitor` defines the `private enum Commands`, symbolically listing the various commands its input language supports, as well as several data members, among which a `Selector` object and a `map` using child order numbers as its keys, and pointer to `Child` objects (see section 20.3.5.3) as its values. Furthermore, `Monitor` has a static array member `s_handler[]`, storing pointers to member functions handling user commands.

A destructor should have been implemented too, but its implementation is left as an exercise to the reader. Before the class interface can be processed by the compiler, it must have seen `select.h` and `child.h`. Here is the class header, including the interface of the nested function object class `Find`:

```
class Monitor
{
    enum Commands
    {
        UNKNOWN,
        START,
        EXIT,
        STOP,
        TEXT,
        sizeofCommands
    };
```

```
    class Find
    {
        int     d_nr;
        public:
            Find(int nr);
            bool operator()(std::map<int, Child *>::value_type &vt)
                                                                const;
    };

    Selector                d_selector;
    int                     d_nr;
    std::map<int, Child *>  d_child;

    static void (Monitor::*s_handler[])(int, std::string const &);

    public:
        enum Done
        {};

        Monitor();
        void run();

    private:
        static void killChild(std::map<int, Child *>::value_type it);
        static void initialize();

        Commands    next(int *value, std::string *line);
        void    processInput();
        void    processChild(int fd);

        void    createNewChild(int, std::string const &);
        void    exiting(int = 0, std::string const &msg = std::string());
        void    sendChild(int value, std::string const &line);
        void    stopChild(int value, std::string const &);
        void    unknown(int, std::string const &);
};
```

Since there's only one non-class type data member, the class's constructor remains very short and could be implemented inline. However, the array `s_handler`, storing pointers to functions needs to be initialized as well. This can be accomplished in several ways:

- Since the `Command` enumeration only contains a fairly limited set of commands, compile-time initialization could be considered:

```
    void (Monitor::*Monitor::s_handler[])(int, string const &)  =
    {
        &Monitor::unknown,              // order follows enum Command's
        &Monitor::createNewChild,       // elements
        &Monitor::exiting,
        &Monitor::stopChild,
        &Monitor::sendChild,
    };
```

The advantage of this is that it's simple, and not requiring any run-time effort. The disadvantage is of course relatively complex maintenance. If for some reason `Commads` is modified, `s_handler` must be modified as well. In cases like these, compile-time initialization is a little bit asking for trouble. There is a simple alternative though, which admittedly does take some execution time:

- A static member may be called before the first `Monitor` object is constructed, which initializes the elements of the array explicitly. This has the advantage of robustness against reordering of enumeration values, which is important: enumerations *do* receive modifications during the development cycle of a class. Maintenance is still required if new values are added to the enumeration, but in that case maintenance is required anyway.

- Using a static member that's explicitly called from `main()` may become a burden, or may be considered unacceptable, as it puts an additional responsibility with the software engineer, rather than with the software. It's a matter of taste whether that's a consideration to take seriously or not. If the initialization function is not called, the program will clearly fail and repairing the error caused by not calling the initialization function is easily repaired. If that's considered bad practice, the initialization function may be called from the class constructors as well. The following initialization function used in the current implementation of the class `Monitor`:

```
void (Monitor::*Monitor::s_handler[sizeofCommands])(int, string const &);

void Monitor::initialize()
{
    if (s_handler[UNKNOWN] != 0)    // already initialized
        return;

    s_handler[UNKNOWN] =    &Monitor::unknown;
    s_handler[START] =      &Monitor::createNewChild;
    s_handler[EXIT] =       &Monitor::exiting;
    s_handler[STOP] =       &Monitor::stopChild;
    s_handler[TEXT] =       &Monitor::sendChild;
}
```

Since the initialization function immediately returns if the initialization has already been performed, `Monitor`'s constructor may call the initialization and still defensibly be implemented inline:

```
inline Monitor::Monitor()
:
    d_nr(0)
{
    initialize();
}
```

The core of `Monitor`'s activities are performed by `run()`. It performs the following tasks:

- Initially, the `Monitor` object only listens to its standard input: the set of input file descriptors to which `d_selector` will listen is initialized to `STDIN_FILENO`.

- Then, in a loop `d_selector`'s `wait()` function is called. If input on `cin` is available, it is processed by `processInput()`. Otherwise, the input has arived from a child process. Information sent by children is processed by `processChild()`.

- To prevent *zombies*, the child processes must catch *their* children's termination signals. This will be discussed below (In an earlier version Monitor caught the termination signals. As noted by Ben Simons (ben at mrxfx dot com) this is inappropriate: the child process itself has that responsibility. Thanks, Ben).

Here is run()'s implementation:

```
#include "monitor.ih"

void Monitor::run()
{
    d_selector.addReadFd(STDIN_FILENO);

    while (true)
    {
        cout << "? " << flush;
        try
        {
            d_selector.wait();

            int fd;
            while ((fd = d_selector.readFd()) != -1)
            {
                if (fd == STDIN_FILENO)
                    processInput();
                else
                    processChild(fd);
            }
            cout << "NEXT ...\n";

        }
        catch (char const *msg)
        {
            exiting(1, msg);
        }
    }
}
```

The member function processInput() reads the commands entered by the user via the program's standard input stream. The member itself is rather simple: it calls next() to obtain the next command entered by the user, and then calls the corresponding function using the matching element of the s_handler[] array. The members processInput() and next() were defined as follows:

```
void Monitor::processInput()
{
    string line;
    int    value;
    Commands cmd = next(&value, &line);
    (this->*s_handler[cmd])(value, line);
}

Monitor::Commands Monitor::next(int *value, string *line)
{
```

```
        if (!getline(cin, *line))
            exiting(1, "Command::next(): reading cin failed");

        if (*line == "start")
            return START;

        if (*line == "exit" || *line == "quit")
        {
            *value = 0;
            return EXIT;
        }

        if (line->find("stop") == 0)
        {
            istringstream istr(line->substr(4));
            istr >> *value;
            return !istr ? UNKNOWN : STOP;
        }

        istringstream istr(line->c_str());
        istr >> *value;
        if (istr)
        {
            getline(istr, *line);
            return TEXT;
        }

        return UNKNOWN;
    }
```

All other input sensed by `d_select` has been created by child processes. Because `d_select`'s `readFd()` member returns the corresponding input file descriptor, this descriptor can be passed to `processChild()`. Then, using a `ifdstreambuf` (see section 20.1.2.1), its information is read from an input stream. The *communication protocol* used here is rather basic: To every line of input sent to a child, the child sends exactly one line of text in return. Consequently, `processChild()` just has to read one line of text:

```
    void Monitor::processChild(int fd)
    {
        ifdstreambuf ifdbuf(fd);
        istream istr(&ifdbuf);
        string line;

        getline(istr, line);
        cout << d_child[fd]->pid() << ": " << line << endl;
    }
```

Please note the construction `d_child[fd]->pid()` used in the above source. `Monitor` defines the data member `map<int, Child *> d_child`. This map contains the child's order number as its key, and a pointer to the `Child` object as its value. A pointer is used here, rather than a `Child` object, since we do want to use the facilities offered by the map, but don't want to copy a `Child` object.

The implication of using pointers as `map`-values is of course that the responsibility to destruct the

Child object once it becomes superfluous now lies with the programmer, and not any more with the run-time support system.

Now that run()'s implementation has been covered, we'll concentrate on the various commands users might enter:

- When the start command is issued, a new child process is started. A new element is added to d_child by the member createNewChild(). Next, the Child object should start its activities, but the Monitor object can not wait here for the child process to complete its activities, as there is no well-defined endpoint in the near future, and the user will probably want to enter more commands. Therefore, the Child process will run as a *daemon*: its parent process will terminate immediately, and its own child process will continue in the background. Consequently, createNewChild() calls the child's fork() member. Although it is the child's fork() function that is called, it is still the monitor program wherein fork() is called. So, the *monitor* program is duplicated by fork(). Execution then continues:
  - At the Child's parentProcess() in its parent process;
  - At the Child's childProcess() in its child process

  As the Child's parentProcess() is an empty function, returning immediately, the Child's parent process effectively continues immediately below createNewChild()'s cp->fork() statement. As the child process never returns (see section 20.3.5.3), the code below cp->fork() is never executed by the Child's child process. This is exactly as it should be.

  In the parent process, createNewChild()'s remaining code simply adds the file descriptor that's available for reading information from the child to the set of input file descriptors monitored by d_select, and uses d_child to establish the association between that file descriptor and the Child object's address:

```
void Monitor::createNewChild(int, string const &)
{
    Child *cp = new Child(++d_nr);

    cp->fork();

    int fd = cp->readFd();

    d_selector.addReadFd(fd);
    d_child[fd] = cp;

    cerr << "Child " << d_nr << " started\n";
}
```

- Direct communication with the child is required for the stop <nr> and <nr> text commands. The former command terminates child process <nr>, by calling stopChild(). This function locates the child process having the order number using an anonymous object of the class Find, nested inside Monitor. The class Find simply compares the provided nr with the children's order number returned by their nr() members:

```
inline Monitor::Find::Find(int nr)
:
    d_nr(nr)
{}
inline bool Monitor::Find::operator()(
                        std::map<int, Child *>::value_type &vt) const
{
```

```
        return d_nr == vt.second->nr();
    }
```

If the child process having order number `nr` was found, its file descriptor is removed from `d_selector`'s set of input file descriptors. Then the child process itself is terminated by the static member `killChild()`. The member `killChild()` is declared as a *static* member function, as it is used as function argument of the `for_each()` generic algorithm by `erase()` (see below). Here is `killChild()`'s implementation:

```
    void Monitor::killChild(map<int, Child *>::value_type it)
    {
        if (kill(it.second->pid(), SIGTERM))
            cerr << "Couldn't kill process " << it.second->pid() << endl;
    }
```

Having terminated the specified child process, the corresponding `Child` object is destroyed and its pointer is removed from `d_child`:

```
    void Monitor::stopChild(int nr, string const &)
    {
        map<int, Child *>::iterator it =
            find_if(d_child.begin(), d_child.end(), Find(nr));

        if (it == d_child.end())
            cerr << "No child number " << nr << endl;
        else
        {
            d_selector.rmReadFd(it->second->readFd());

            delete it->second;
            d_child.erase(it);
        }
    }
```

- The command `<nr> text>` will send `text` to child process `nr`, using the member function `sendChild()`. This function too, will use a `Find` object to locate the process having order number `nr`, and will then simply insert the text into the writing end of a pipe connected to the indicated child process:

```
    void Monitor::sendChild(int nr, string const &line)
    {
        map<int, Child *>::iterator it =
            find_if(d_child.begin(), d_child.end(), Find(nr));

        if (it == d_child.end())
            cerr << "No child number " << nr << endl;
        else
        {
            ofdnstreambuf ofdn(it->second->writeFd());
            ostream out(&ofdn);

            out << line << endl;
        }
    }
```

- When users enter `exit` the member `exiting()` is called. It terminates all child processes, by visiting all elements of `d_child`, using the `for_each()` generic algorithm (see section 17.4.17). The program is subsequently terminated:

```
void Monitor::exiting(int value, string const &msg)
{
    for_each(d_child.begin(), d_child.end(), killChild);
    if (msg.length())
        cerr << msg << endl;
    throw value;
}
```

Finally, the program's `main()` function is simply:

```
#include "monitor.h"

int main()
try
{
    Monitor monitor;

    monitor.run();
}
catch (int exitValue)
{
    return exitValue;
}

/*
    Example of a session:

    # a.out
    ? start
    Child 1 started
    ? 1 hello world
    ? 3394: Child 1:1:  hello world
    ? 1 hi there!
    ? 3394: Child 1:2:  hi there!
    ? start
    Child 2 started
    ? 3394: Child 1: standing by
    ? 3395: Child 2: standing by
    ? 3394: Child 1: standing by
    ? 3395: Child 2: standing by
    ? stop 1
    ? 3395: Child 2: standing by
    ? 2 hello world
    ? 3395: Child 2:1:  hello world
    ? 1 hello world
    No child number 1
    ? exit3395: Child 2: standing by
    ?
    #
*/
```

20.3.5.3 The class 'Child'

When the Monitor object starts a child process, it has to create an object of the class Child. The Child class is derived from the class Fork, allowing its construction as a *daemon*, as discussed in the previous section. Since a Child object is a daemon, we know that its parent process should be defined as an empty function. its childProcess() must of course still be defined. Here are the characteristics of the class Child:

- The Child class defines two Pipe data members, to allow communications between its own child- and parent processes. As these pipes are used by the Child's child process, their names are aimed at the child process: the child process reads from d_in, and writes to d_out. Here is the interface of the class Child:

```
class Child: public Fork
{
    Pipe                d_in;
    Pipe                d_out;

    int         d_parentReadFd;
    int         d_parentWriteFd;
    int         d_nr;

    public:
        Child(int nr);
        virtual ~Child();
        int readFd() const;
        int writeFd() const;
        int pid() const;
        int nr() const;
        virtual void childRedirections();
        virtual void parentRedirections();
        virtual void childProcess();
        virtual void parentProcess();
};
```

- The Child's constructor simply stores its argument, a child-process order number, in its own d_nr data member:

```
inline Child::Child(int nr)
:
    d_nr(nr)
{}
```

- The Child's child process will simply obtain its information from its standard input stream, and it will write its information to its standard output stream. Since the communication channels are pipes, redirections must be configured. The childRedirections() member is implemented as follows:

```
void Child::childRedirections()
{
    d_in.readFrom(STDIN_FILENO);
    d_out.writtenBy(STDOUT_FILENO);
}
```

- Although the parent process performs no actions, it must configure some redirections. Since the names of the pipes indicate their functions in the child process, `d_in` is used for *writing* by the parent, and `d_out` is used for *reading* by the parent. Here is the implementation of `parentRedirections()`:

```
void Child::parentRedirections()
{
    d_parentReadFd = d_out.readOnly();
    d_parentWriteFd = d_in.writeOnly();
}
```

- The `Child` object will exist until it is destroyed by the `Monitor`'s `stopChild()` member. By allowing its creator, the `Monitor` object, to access the parent-side ends of the pipes, the `Monitor` object can communicate with the `Child`'s child process via those pipe-ends. The members `readFd()` and `writeFd()` allow the `Monitor` object to access these pipe-ends:

```
inline int Child::readFd() const
{
    return d_parentReadFd;
}
inline int Child::writeFd() const
{
    return d_parentWriteFd;
}
```

- The `Child` object's child process basically has two tasks to perform:
  - It must reply to information appearing at its standard input stream;
  - If no information has appeared within a certain time frame (the implementations uses an interval of five seconds), then a message should be written to its standard output stream anyway.

  To implement this behavior, `childProcess()` defines a local `Selector` object, adding `STDIN_FILENO` to its set of monitored input file descriptors.

  Then, in an eternal loop, `childProcess()` waits for `selector.wait()` to return. When the alarm goes off, it sends a message to its standard output. (Hence, into the writing pipe). Otherwise, it will echo the messages appearing at its standard input to its standard output. Here is the implementation of the `childProcess()` member:

```
void Child::childProcess()
{
    Selector    selector;
    size_t    message = 0;

    selector.addReadFd(STDIN_FILENO);
    selector.setAlarm(5);

    while (true)
    {
        try
        {
            if (!selector.wait())       // timeout
                cout << "Child " << d_nr << ": standing by\n";
            else
            {
```

```
                string  line;
                getline(cin, line);
                cout << "Child " << d_nr << ":" << ++message << ": " <<
                                                    line << endl;
            }
        }
        catch (...)
        {
                cout << "Child " << d_nr << ":" << ++message << ": " <<
                            "select() failed" << endl;
        }
    }
    exit(0);
}
```

- Next, twoaccessors allow the `Monitor` object to obtain the `Child`'s process ID and order number, respectively:

```
inline int Child::pid() const
{
    return Fork::pid();
}
inline int Child::nr() const
{
    return d_nr;
}
```

- A `Child` process terminates when the user enters a `stop` command. When an existing child process number was entered, the corresponding `Child` object is removed from `Monitor`'s `d_child` map. As a result, its destructor is called. In its turn, `Child`'s destructor will call `kill` to terminate its child, and then waits for the child to terminate. Once the child has terminated, the destructor has completed its work as well and returns, competing the erasure from `d_child`. The implementation offered here will fail if the child process doesn't react to the `SIGTERM` signal. In this demonstration program this does not happen. In 'real life' implementations more elaborate killing-procedures may be required (e.g., using `SIGKILL` in addition to `SIGTERM`). As discussed in section 8.8 it is important to ensure that the destruction succeeds. Here is the implementation of the `Child`'s destructor:

```
Child::~Child()
{
    if (pid())
    {
        cout << "Killing process " << pid() << "\n";
        kill(pid(), SIGTERM);
        int status;
        wait(&status);
    }
}
```

20.4 Function objects performing bitwise operations

In section 17.1 several types of predefined function objects were introduced. Predefined function objects performing arithmetic operations, relational operations, and logical operations exist, corresponding to a multitude of binary- and unary operators.

Some operators appear to be missing: there appear to be no predefined function objects corresponding to bitwise operations. However, their construction is, given the available predefined function objects, not difficult. The following examples show a template class implementing a function object calling the bitwise and (`operator&()`), and a template class implementing a function object calling the unary not (`operator~()`). It is left to the reader to construct similar function objects for other operators.

Here is the implementation of a function object calling the bitwise `operator&()`:

```
#include <functional>

template <typename _Tp>
struct bit_and: public std::binary_function<_Tp, _Tp, _Tp>
{
    _Tp operator()(_Tp const &__x, _Tp const &__y) const
    {
        return __x & __y;
    }
};
```

Here is the implementation of a function object calling `operator~()`:

```
#include <functional>

template <typename _Tp>
struct bit_not: public std::unary_function<_Tp, _Tp>
{
    _Tp operator()(_Tp const &__x) const
    {
        return ~__x;
    }
};
```

These and other missing predefined function objects are also implemented in the file `bitfunctional`, which is found in the `cplusplus.yo.zip` archive. It should be noted that these classes are derived from existing template classes (e.g., `std::binary_function` and `std::unary_function`). These base classes offer several typedefs which are expected (used) by various generic algorithms as defined in the STL (cf. chapter 17), thus following the advice offered in, e.g., the **C++** header file `bits/stl_function.h`:

```
 *  The standard functors are derived from structs named unary_function
 *  and binary_function.  These two classes contain nothing but typedefs,
 *  to aid in generic (template) programming.  If you write your own
 *  functors, you might consider doing the same.
```

Here is an example using `bit_and()` removing all odd numbers from a vector of `int` values:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include "bitand.h"
using namespace std;
```

```
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
*/
```

20.5 Implementing a 'reverse_iterator'

Earlier, in section 19.12.1, the construction of iterators and reverse iteraters was discussed. In that section the iterator was constructed as an inner class in a class derived from a vector of pointers to strings.

An object of this nested iterator class handled the dereferencing of the pointers stored in the vector. This allowed us to sort the *strings* pointed to by the vector's elements rather than the *pointers*.

A drawback of the approach taken in section 19.12.1 is that the class implementing the iterator is closely tied to the derived class as the iterator class was implemented as a nested class. What if we would like to provide any class derived from a container class storing pointers with an iterator handling the pointer-dereferencing?

In this section a variant to the earlier (nested class) approach is discussed. The iterator class will be defined as a *template class*, parameterizing the data type to which the container's elements point as well as the iterator type of the container itself. Once again, we will implement a *RandomIterator* as it is the most complex iterator type.

Our class is named `RandomPtrIterator`, indicating that it is a random iterator operating on pointer values. The template class defines three template type parameters:

- The first parameter specifies the derived class type (`Class`). Like the earlier nested class, `RandomPtrIterator`'s constructor will be private. Therefore we need `friend` declarations to allow client classes to construct `RandomPtrIterators`. However, a `friend class Class` cannot be defined: template parameter types cannot be used in `friend class ...` declarations. But this is no big problem: not every member of the client class needs to construct iterators. In fact, only `Class`'s `begin()` and `end()` members must be able to construct iterators. Using the template's first parameter, friend declarations can be specified for the client's `begin()` and `end()` members.

- The second template parameter parameterizes the container's iterator type (`BaseIterator`);

- The third template parameter indicates the data type to which the pointers point (`Type`).

`RandomPtrIterator` uses one private data element, a `BaseIterator`. Here is the class interface, including the constructor's implementation:

```
#include <iterator>

template <typename Class, typename BaseIterator, typename Type>
class RandomPtrIterator:
      public std::iterator<std::random_access_iterator_tag, Type>
{
    friend RandomPtrIterator<Class, BaseIterator, Type> Class::begin();
    friend RandomPtrIterator<Class, BaseIterator, Type> Class::end();

    BaseIterator d_current;

    RandomPtrIterator(BaseIterator const &current);

    public:
        bool operator!=(RandomPtrIterator const &other) const;
        int operator-(RandomPtrIterator const &rhs) const;
        RandomPtrIterator const operator+(int step) const;
        Type &operator*() const;
        bool operator<(RandomPtrIterator const &other) const;
        RandomPtrIterator &operator--();
        RandomPtrIterator const operator--(int);
        RandomPtrIterator &operator++();
        RandomPtrIterator const operator++(int);
        bool operator==(RandomPtrIterator const &other) const;
        RandomPtrIterator const operator-(int step) const;
        RandomPtrIterator &operator-=(int step);
        RandomPtrIterator &operator+=(int step);
        Type *operator->() const;
};

template <typename Class, typename BaseIterator, typename Type>
RandomPtrIterator<Class, BaseIterator, Type>::RandomPtrIterator(
                                BaseIterator const &current)
:
    d_current(current)
{}
```

Dissecting its `friend` declarations, we see that the members `begin()` and `end()` of a class `Class`, returning a `RandomPtrIterator` object for the types `Class, BaseIterator` and `Type` are granted access to `RandomPtrIterator`'s private constructor. That is exactly what we want. Note that `begin()` and `end()` are declared as *bound friends*.

All `RandomPtrIterator`'s remaining members are public. Since `RandomPtrIterator` is just a generalization of the nested class `iterator` developed in section 19.12.1, re-implementing the required member functions is easy, and only requires us to change `iterator` into `RandomPtrIterator` and to change `std::string` into `Type`. For example, `operator<()`, defined in the class `iterator` as

```
inline bool StringPtr::iterator::operator<(iterator const &other) const
```

```
{
    return **d_current < **other.d_current;
}
```

is re-implemented as:

```
    template <typename Class, typename BaseIterator, typename Type>
    bool RandomPtrIterator<Class, BaseIterator, Type>::operator<(
                                RandomPtrIterator const &other) const
    {
        return **d_current < **other.d_current;
    }
```

As a second example: `operator*()`, defined in the class `iterator` as

```
inline std::string &StringPtr::iterator::operator*() const
{
    return **d_current;
}
```

is re-implemented as:

```
    template <typename Class, typename BaseIterator, typename Type>
    Type &RandomPtrIterator<Class, BaseIterator, Type>::operator*() const
    {
        return **d_current;
    }
```

The pre- and postfix increment operators are re-implemented as:

```
    template <typename Class, typename BaseIterator, typename Type>
    RandomPtrIterator<Class, BaseIterator, Type>
    &RandomPtrIterator<Class, BaseIterator, Type>::operator++()
    {
        ++d_current;
        return *this;
    }
    template <typename Class, typename BaseIterator, typename Type>
    RandomPtrIterator<Class, BaseIterator, Type> const
    RandomPtrIterator<Class, BaseIterator, Type>::operator++(int)
    {
        return RandomPtrIterator(d_current++);
    }
```

Remaining members can be implemented accordingly, their actual implementations are left as an exercise to the reader (or can be obtained from the `cplusplus.yo.zip` archive, of course).

Reimplementing the class `StringPtr` developed in section 19.12.1 is not difficult either. Apart from including the header file defining the template class `RandomPtrIterator`, it requires only a single modification as its `iterator` typedef must now be associated with a `RandomPtrIterator`. Here are the full class interface and inline member definitions:

```
    #ifndef _INCLUDED_STRINGPTR_H_
```

```
#define _INCLUDED_STRINGPTR_H_

#include <vector>
#include <string>
#include "iterator.h"

class StringPtr: public std::vector<std::string *>
{
    public:
        typedef RandomPtrIterator
                <
                    StringPtr,
                    std::vector<std::string *>::iterator,
                    std::string
                >
                    iterator;

        typedef std::reverse_iterator<iterator> reverse_iterator;

        iterator begin();
        iterator end();
        reverse_iterator rbegin();
        reverse_iterator rend();
};

inline StringPtr::iterator StringPtr::begin()
{
    return iterator(this->std::vector<std::string *>::begin() );
}
inline StringPtr::iterator StringPtr::end()
{
    return iterator(this->std::vector<std::string *>::end());
}
inline StringPtr::reverse_iterator StringPtr::rbegin()
{
    return reverse_iterator(end());
}
inline StringPtr::reverse_iterator StringPtr::rend()
{
    return reverse_iterator(begin());
}
#endif
```

Including `StringPtr`'s modified header file into the program given in section 19.12.2 will result in a program behaving identically to its earlier version, albeit that `StringPtr::begin()` and `StringPtr::end()` now return iterator objects constructed from a template definition.

20.6 A text to anything converter

The standard **C** library offers conversion functions like `atoi()`, `atol()`, and other functions, which can be used to convert `ASCII-Z` strings to numerical values. In **C++**, these functions are still available, but a more *type safe* way to convert text to other types is by using objects of the class `std::istringsteam`.

Using the `std::istringstream` class instead of the **C** standard conversion functions may have the advantage of type-safety, but it also appears to be a rather cumbersome alternative. After all, we will have to construct and initialize a `std::istringstream` object first, before we're actually able to extract a value of some type from it. This requires us to use a variable. Then, if the extracted value is actually only needed to initialize some function-parameter, one might wonder whether the additional variable and the `istringstream` construction can somehow be avoided.

In this section we'll develop a class (`A2x`) preventing all the disadvantages of the standard **C** library functions, without requiring the cumbersome definitions of `std::istringstream` objects over and over again. The class is called `A2x` for 'ascii to anything'.

`A2x` objects can be used to obtain a value for any type extractable from `std::istream` objects given its textual representation. Since `A2x` represents the object-variant of the **C** functions, it is not only type-safe but *also* extensible. Consequently, their use is greatly preferred over the standard **C** functions. Here are its characteristics:

- `A2x` is derived from `std::istringstream`, so all members of the class `std::istringstream` are available. Thus, extractions of values of variables can always be performed effortlessly. Here's the class's interface:

```
class A2x: public std::istringstream
{
    public:
        A2x();
        A2x(char const *txt);
        A2x(std::string const &str);

        template <typename T>
        operator T();

        A2x &operator=(char const *txt);

        A2x &operator=(std::string const &str);
        A2x &operator=(A2x const &other);
};
```

- `A2x` has a default constructor and a constructor expecting a `std::string` argument. The latter constructor may be used to initialize `A2x` objects with text to be converted (e.g., a line of text obtained from reading a configuration file):

```
inline A2x::A2x()
{}

inline A2x::A2x(char const *txt)                    // initialize from text
:
    std::istringstream(txt)
{}

inline A2x::A2x(std::string const &str)
:
    std::istringstream(str.c_str())
{}
```

- A2x's real strength comes from its `operator Type()` conversion member template. As it is a member template, it will automatically adapt itself to the type of the variable that should be given a value, obtained by converting the text stored inside the A2x object to the variable's type. When the extraction fails, A2x's inherited `good()` member will return `false`:

  ```
  template <typename Type>
  inline A2x::operator Type()
  {
      Type t;

      return (*this >> t) ? t : Type();
  }
  /=
  inline A2x &A2x::operator=(std::string const &str)
  {
      return operator=(str.c_str());
  }
  //OP=
  inline A2x &A2x::operator=(A2x const &other)
  {
      return operator=(other.str());
  }
  ```

- Occasionally, the compiler may not be able to determine which type to convert to. In that case, an *explicit template type* can be used:

  ```
  A2x.operator int<int>();
  // or just:
  A2x.operator int();
  ```

  Since neither syntax looks attractive, the member template `to()` was provided as well, allowing constructions like:

  ```
  A2x.to(int());
  ```

  Here is its implementation:

- Once an A2x object is available, it may be reinitialized using its `operator=()` member:

  ```
  #include "a2x.h"

  A2x &A2x::operator=(std::string const &txt)
  {
      clear();        // very important!!! If a conversion failed, the object
                      // remains useless until executing this statement
      str(txt);
      return *this;
  }
  ```

Here are some examples of its use:

```
int x = A2x("12");          // initialize int x from a string "12"
A2x a2x("12.50");           // explicitly create an A2x object
```

```
double d;
d = a2x;                     // assign a variable using an A2x object

a2x = "err";
d = a2x;                     // d is 0: the conversion failed,
                             // and a2x.good() == false

a2x = " a";                  // reassign a2x to new text
char c = a2x;                // c now 'a': internally operator>>() is used
                             // so initial blanks are skipped.

extern expectsInt(int x);    // initialize a parameter using an
expectsInt(A2x("1200"));     // anonymous A2x object

d = A2x("12.45").to(int()); // d is 12, not 12.45
```

Apart from a class `A2x` a complementary class (`X2a`) can easily be constructed as well. The construction of `X2a` is left as an exercise to the reader.

20.7 Wrappers for STL algorithms

Many generic algorithms (cf. chapter 17) use function objects to operate on the data to which their iterators refer, or they require predicate function objects using some criterion to make a decision about these data. The standard approach followed by the generic algorithms is to pass the information to which the iterators refer to overloaded function call operators (i.e., `operator()()`) of function objects that are passed as arguments to the generic algorithms.

Usually this approach requires the construction of a dedicated class implementing the required function object. However, in many cases the *class context* in which the iterators exist already offers the required functionality. Alternatively, the functionality might exist as member function of the objects to which the iterators refer. For example, finding the first empty `string` object in a vector of `string` objects could profitably use the `string::empty()` member.

Another frequently encountered situation is related to a *local context*. Once again, consider the situation where the elements of a `string` vector are all visited: each object must be inserted in a stream whose reference is only known to the function in which the `string` elements are visited, but some additional information must be passed to the insertion function as well, making the use of the `ostream_inserter` less appropriate.

The frustrating part of using generic algorithms is that these dedicated function objects often very much look like each other, but the standard solution (using predefined function objects, using specialized iterators) seldomly do the required job: their fixed function interfaces (e.g., `equal_to` calling the object's `operator==()`) often are too rigid to be useful and, furthermore, they are unable to use any additional local context that is active when they are used.

Nevertheless, one may wonder whether template classes might be constructed which can be used again and again to create dedicated function objects. Such template class instantiations should offer facilities to call configurable (member) functions, using a configurable local context.

In the upcoming sections, several *wrapper templates* supporting these requirements are developed. To support a *local context*, a dedicated *local context struct* is introduced. Furthermore, the wrapper templates will allow us to specify the member function that should be called in its constructor. Thus the rigidness of the fixed member function as used in the predefined function objects is avoided.

As an example of a generic algorithm usually requiring a simple function object, consider `for_each()`. The `operator()()` of the function object passed to this algorithm receives as its argument a reference to the object to which the iterators refer. Generally, the `operator()()` will do one of two things:

- It may call a member function of the object defined in its parameter list (e.g., `operator()(string &str)` may call `str.length()`);
- It may call a function, passing it its parameter as argument (e.g., calling `somefunction(str)`).

Of course, the latter example is a bit overkill, since `somefunction()`'s address could actually directly have been passed to the generic algorithm, so why use this complex procedure? The answer is *context*: if `somefunction()` would actually require other arguments, representing the local context in which `somefunction()` was called, then the function object's constructor could have received the local context as its arguments, passing that local context on to `somefunction()`, together with the object received by the function object's `operator()()` function. There is no way to pass any local context to the generic algorithm's simple variant, in which a function's address is passed to the generic function.

At first sight, however, the fact that a local context differs from one situation to another makes it hard to standardize the local context: a local context might consist of values, pointers, references, which differ in number and types from one situation to another. Defining templates for all possible situations is clearly impractical, and using **C**-style variadic functions is also not very attractive, since the arguments passed to a variadic function object constructor cannot simply be passed on to the function object's `operator()()`.

The concept of a *local context struct* is introduced to standardize the local context. It is based on the following considerations:

- Usually, a function requiring a local context is a member function of some class.
- Instead of using the intuitive implementation where the member function is given the required parameters representing a local context, it receives a single argument: a `const &` to a local context `struct`.
- The local context `struct` is defined in the function's class interface.
- Before the function is called, a local context `struct` is initialized, which is then passed as argument to the function.

Of course, the organization of local context `structs` will differ from one situation to the next situation, but there is always just *one* local context required. The fact that the inner organization of the local context differs from one situation to the next causes no difficulty at all to **C++**'s template mechanism. Actually, having available a generic type (*Context*) together with several concrete instantiations of that generic type is a mere text-book argument for using templates.

20.7.1 Local context structs

When a function is called, the context in which it is called is made known to the function by providing the function with a parameter list. When the function is called, these parameters are initialized by the function's arguments. For example, a function `show()` may expect two arguments: an `ostream &` into which the information is inserted and an object which will be inserted into the stream. For example:

```
void State::show(ostream &out, Item const &item)
```

```
{
    out << "Here is item " << item.nr() << ":\n" <<
            item << endl;
}
```

Functions clearly differ in their parameter lists: both the numbers and types of their parameters vary.

A *local context struct* is used to standardize the parameter lists of functions, for the benefit of template construction. In the above example, the function `State::show()` uses a local context consisting of an `ostream &` and an `Item const &`. This context never changes, and may very well be offered through a `struct` defined as follows:

```
struct ShowContext
{
    ostream &out;
    Item const &item;
};
```

Note that this `struct` mimics `State::show()`'s parameter list. Since it is directly connected to the function `State::show()` it is best defined in the class `State`, offering the function `State::show()`. Once we have defined this `struct`, `State::show()`'s implementation is modified so that it now expects a `ShowContext &`:

```
void State::show(ShowContext &context)
{
    context.out << "Here is item " << context.item.nr() << ":\n" <<
            context.item << endl;
}
```

(Alternatively, an overloaded `State::show(ShowContext &context)` could be defined, calling the original `show()` member).

Using a local context `struct` any parameter list (except those of variadic functions) can be standardized to a parameter list consisting of a single element. Now that we have a single parameter to specify any local context we're ready for the 'templatization' of function object wrapper classes.

20.7.2 Member functions called from function objects

The member function called by function objects is the function `operator()()`, which may be defined as a function having various parameters. In the context of generic algorithms, these parameters are usually one or two elements, representing the data to which the algorithm's iterators point. Unfortunately from the point of view of the template class constructor, it is not known beforehand whether these data elements are objects, primitive types, or pointers. Let's assume that we would like to create a function object changing all letters in `string` objects into capital letters. In that case our `operator()()` function may receive a `string &` (e.g., when iterating over the elements of a `vector<string>`), but our `operator()()` function may also receive a `string *` (e.g., when iterating over the elements of a `vector<string *>`). Other parameter types can be conceived of as well.

So, how can we define a generic function that can be called from `operator()()` if we don't know (when defining the template) whether we should call using `.*` or `->*`? The issue whether to call a member function using a pointer to member in combination with an object or a pointer to object does not have to be solved by the template. Instead it can be handled by the class itself, if the class provides an appropriate *static* member.

An additional advantage of using a static function is that the static members do not have `const` attributes. Consequently, no ambiguity can arise when calling a static member function from within a function object's `operator()()`.

Generic algorithms, however, differ in their using of the function object's `operator()()`'s return value. As will be illustrated in the next section, the return type of called functions may also be parameterized.

20.7.3 The configurable, single argument function object template

As an introductory example, let's assume we have a class `Strings` holding a `vector<string> d_vs` data member. We would like to change all letter-characters in the strings stored in `d_vs` into upper case characters, and we would like to insert the original and modified strings into a configurable `ostream` object. To accomplish this, our class offers a member `uppercase(ostream &out)`.

We would like to use the `for_each()` generic algorithm. This algorithm may be given a function's address, or it may be given a function object. Clearly, since we have a local context (the configurable `ostream` object), the function object is required here. Therefore, the following support class is constructed:

```
class Support
{
    std::ostream &d_out;

    public:
        Support(std::ostream &out);
        void operator()(std::string &str) const;
};

inline Support::Support(std::ostream &out)
:
    d_out(out)
{}

inline void Support::operator()(std::string &str) const
{
    d_out << str << " ";
    transform(str.begin(), str.end(), str.begin(), toupper);
    d_out << str << std::endl;
}
```

An anonymous `Support` class object may now be used in the implementation of the class `Strings`. Here is an example of its definition and use:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
```

```
#include "support.h"

class Strings
{
    std::vector<std::string> d_vs;

    public:
        void uppercase(std::ostream &out);
};

void Strings::uppercase(std::ostream &out)
{
    for_each(d_vs.begin(), d_vs.end(), Support(out));
}

using namespace std;

int main()
{
    Strings s;

    s.uppercase(cout);
}
```

To 'templatize' the `Support` class, using the considerations discussed previously, we perform the following steps:

- The local context will be put in a `struct`, which is then passed to the template's constructor, so `Context` becomes one of the template type parameters.
- The implementation of the template's `operator()()` is standardized. In the template it will call a function, receiving the `operator()()`'s argument (which also becomes a template parameter) and a reference to the context as its arguments. The address of the function to call may be stored in a local variable of the template function object. In the `Support` class, `operator()()` uses a `void` return type. This type is often the required type, but when defining predicates it may be a `bool`. Therefore, the return type of the template's `operator()()` (and thus the return type of the called function) is made configurable as well, offering a default type `void` for convenience. Thus, we get the following definition of the variable holding the address of the function to call:

  ```
  ReturnType (*d_fun)(Type &argument, Context &context);
  ```

  and the template's `operator()()` implementation (passing it another template data member: `Context &d_context`) becomes:

  ```
  template<typename Type, typename Context, typename ReturnType>
  ReturnType Wrap1<Type, Context, ReturnType>::operator()(
                                              Type &param) const
  {
      return (*d_fun)(param, d_context);
  }
  ```

- The template's constructor is given two parameters: a function address and a reference to the local context `struct`. Coining the classname `Wrap1` (for unary (1) function object wrapper), its implementation becomes:

```
template<typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::
Wrap1(ReturnType (*fun)(Type &, Context &), Context &context)
:
    d_fun(fun),
    d_context(context)
{}
```

Now we're almost ready to construct the full template class `Wrap1`. Two additional situations need further consideration:

- Arguments passed to the template's `operator()()` member may be of various kinds: values, modifiable references, immutable (`const`) references, pointers to modifiable entities or pointers to immutable entities. The template should offer facilities to use all these different argument types.
- Algorithms defined in the standard template library, notably those requiring *predicate* function objects (e.g., `find_if()`), assume that these objects define internal types, named `result_type` for its `operator()()` member, and `argument_type` for its data type. With binary predicate function objects (see section 20.7.4) `first_argument_type` and `second_argument_type` for the respective types of its `operator()()`'s arguments are expected. Moreover, these types must be 'plain' type names, no pointers nor references.

Various parameter types of the template's `operator()()` function may be handled by overloaded versions of both the template constructor and its `operator()()` member, defining four implementations handling `Type const` references and `Type const` pointers. For each of these situations a function pointer to a corresponding function, called by the template's `operator()()` must be defined as well. Since in each instantiation of the template only *one* type of the overloaded functions (constructor and associated `operator()()`) will be used, a `union` can be defined accomodating the pointers to the various (i.e., four) types of functions that may be passed to the template's constructor. This `union` may be *anonymous*, as only its fields will be used. Note that *value* arguments may be handled by `Type const &` parameters: no additional overloaded version is required to handle value-type arguments.

The internal types expected by some of the STL functions can simply be made available by defining internal `typedefs`. Since the various types of arguments (`const`, pointers, references) are handled by the template's overloaded constructors and member functions, the typedefs may simply set up aliases for the template parameter types.

Here is the full implementation of the configurable, single argument function object template:

```
template <typename Type, typename Context, typename ReturnType = void>
class Wrap1
{
    union
    {
        Context *d_context;
        Context const *d_contextconst;
    };
    union
    {
        ReturnType (*d_ref)(Type &, Context &);
        ReturnType (*d_refPtr)(Type &, Context *);
        ReturnType (*d_refCref)(Type &, Context const &);
```

```
        ReturnType (*d_refCptr)(Type &, Context const *);

        ReturnType (*d_ptr)(Type *, Context &);
        ReturnType (*d_ptr2)(Type *, Context *);
        ReturnType (*d_ptrCref)(Type *, Context const &);
        ReturnType (*d_ptrCptr)(Type *, Context const *);

        ReturnType (*d_crefRef)(Type const &, Context &);
        ReturnType (*d_crefPtr)(Type const &, Context *);
        ReturnType (*d_cref2)(Type const &, Context const &);
        ReturnType (*d_crefCptr)(Type const &, Context const *);

        ReturnType (*d_cptrRef)(Type const *, Context &);
        ReturnType (*d_cptrPtr)(Type const *, Context *);
        ReturnType (*d_cptrCref)(Type const *, Context const &);
        ReturnType (*d_cptr2)(Type const *, Context const *);
    };

    public:
        typedef Type        argument_type;
        typedef ReturnType  result_type;


        // Type may be &, *, const &, and const *
        // Context may be &, *, const &. and const *
        // This allows for 16 combinations which are now all implemented
        // below

        Wrap1(ReturnType (*fun)(Type &, Context &), Context &context);
        Wrap1(ReturnType (*fun)(Type &, Context const &),
               Context const &context);
        Wrap1(ReturnType (*fun)(Type const &, Context &),
               Context &context);
        Wrap1(ReturnType (*fun)(Type const &, Context const &),
               Context const &context);
        Wrap1(ReturnType (*fun)(Type *, Context &),
               Context &context);
        Wrap1(ReturnType (*fun)(Type *, Context const &),
               Context const &context);
        Wrap1(ReturnType (*fun)(Type const *, Context &),
               Context &context);
        Wrap1(ReturnType (*fun)(Type const *, Context const &),
               Context const &context);

        // The following additional constructors are identical to the
        // constructors listed above, but they accept a pointer to a
        // context in various forms.

        Wrap1(ReturnType (*fun)(Type &, Context *), Context *context);
        Wrap1(ReturnType (*fun)(Type &, Context const *),
               Context const *context);
        Wrap1(ReturnType (*fun)(Type const &, Context *),
               Context *context);
        Wrap1(ReturnType (*fun)(Type const &, Context const *),
```

```
                Context const *context);
            Wrap1(ReturnType (*fun)(Type *, Context *), Context *context);
            Wrap1(ReturnType (*fun)(Type *, Context const *),
                Context const *context);
            Wrap1(ReturnType (*fun)(Type const *, Context *),
                Context *context);
            Wrap1(ReturnType (*fun)(Type const *, Context const *),
                Context const *context);

            ReturnType operator()(Type &param) const;
            ReturnType operator()(Type const &param) const;
            ReturnType operator()(Type *param) const;
            ReturnType operator()(Type const *param) const;
    };


template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type &, Context &), Context &context)
:
    d_context(&context),
    d_ref(fun)
{}
                                    // reference Wrap1::const
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type &, Context const &),
                Context const &context)
:
    d_contextconst(&context),
    d_refCref(fun)
{}

                                    // const reference
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type const &, Context &),
                Context &context)
:
    d_context(&context),
    d_crefRef(fun)
{}
                                    // const reference const
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type const &, Context const &),
                Context const &context)
:
    d_contextconst(&context),
    d_cref2(fun)
{}

                                    // pointer
template <typename Type, typename Context, typename ReturnType>
```

```
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type *, Context &),
       Context &context)
:
    d_context(&context),
    d_ptr(fun)
{}
                                    // pointer const
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type *, Context const &),
                Context const &context)
:
    d_contextconst(&context),
    d_ptrCref(fun)
{}

                                    // const pointer
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type const *, Context &),
                Context &context)
:
    d_context(&context),
    d_cptrRef(fun)
{}
                                    // const pointer const
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type const *, Context const &),
                Context const &context)
:
    d_contextconst(&context),
    d_cptrCref(fun)
{}

                                    // reference
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type &, Context *), Context *context)
:
    d_context(context),
    d_refPtr(fun)
{}
                                    // reference const
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type &, Context const *),
                Context const *context)
:
    d_contextconst(context),
    d_refCptr(fun)
{}
```

```
                                    // const reference
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type const &, Context *), Context *context)
:
    d_context(context),
    d_crefPtr(fun)
{}
                                    // const reference const
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type const &, Context const *),
                Context const *context)
:
    d_contextconst(context),
    d_crefCptr(fun)
{}

                                    // pointer
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type *, Context *), Context *context)
:
    d_context(context),
    d_ptr2(fun)
{}
                                    // pointer const
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type *, Context const *),
                Context const *context)
:
    d_contextconst(context),
    d_ptrCptr(fun)
{}
                                    // const pointer
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type const *, Context *), Context *context)
:
    d_context(context),
    d_cptrPtr(fun)
{}
                                    // const pointer const
template <typename Type, typename Context, typename ReturnType>
Wrap1<Type, Context, ReturnType>::Wrap1(
                ReturnType (*fun)(Type const *, Context const *),
                Context const *context)
:
    d_contextconst(context),
    d_cptr2(fun)
{}

template <typename Type, typename Context, typename ReturnType>
```

```
ReturnType Wrap1<Type, Context, ReturnType>::operator()(Type &param) const
{
    return (*d_ref)(param, *d_context);
}

template <typename Type, typename Context, typename ReturnType>
ReturnType
        Wrap1<Type, Context, ReturnType>::operator()(Type const &param) const
{
    return (*d_crefRef)(param, *d_context);
}

template <typename Type, typename Context, typename ReturnType>
ReturnType Wrap1<Type, Context, ReturnType>::operator()(Type *param) const
{
    return (*d_ref)(*param, *d_context);
}

template <typename Type, typename Context, typename ReturnType>
ReturnType
        Wrap1<Type, Context, ReturnType>::operator()(Type const *param) const
{
    return (*d_crefRef)(*param, *d_context);
}
```

To use this template, the original dedicated implementation of `Support::operator()()` is now defined in a static member function of the class `String`, also defining the required local context struct. Here is the new implementation of the class `Strings`, using the template `Wrap1`:

```
    #include <iostream>
    #include <string>
    #include <vector>
    #include <algorithm>

    #include "wrap1.h"

    class Strings
    {
        std::vector<std::string> d_vs;

        struct Context
        {
            std::ostream &out;
        };

        public:
            void uppercase(std::ostream &out);

        private:
            static void xform(std::string &str, Context &context);
    };

    void Strings::uppercase(std::ostream &out)
    {
```

```
    Context context = {out};
    for_each(d_vs.begin(), d_vs.end(),
        Wrap1<std::string, Context>(&xform, context));
}

void Strings::xform(std::string &str, Context &context)
{
    context.out << str << " ";
    transform(str.begin(), str.end(), str.begin(), toupper);
    context.out << str << std::endl;
}

using namespace std;

int main()
{
    Strings s;

    s.uppercase(cout);
}
```

To illustrate the use of the `ReturnType` template parameter, let's assume that the transformations are only required up to the first empty string. In this case, the `find_if` generic algorithm comes in handy, since it stops once a predicate returns `true`. The `xform()` function should return a `bool` value, and the `uppercase()` implementation specifies an explicit type (`bool`) for the `ReturnType` template parameter:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

#include "wrap1.h"

class Strings
{
    std::vector<std::string> d_vs;

    struct Context
    {
        std::ostream &out;
    };

    public:
        void uppercase(std::ostream &out);

    private:
        static bool xform(std::string &str, Context &context);
};

void Strings::uppercase(std::ostream &out)
{
    Context context = {out};
    find_if(d_vs.begin(), d_vs.end(),
```

```
            Wrap1<std::string, Context, bool>(&xform, context));
    }

    bool Strings::xform(std::string &str, Context &context)
    {
        context.out << str << " ";
        transform(str.begin(), str.end(), str.begin(), toupper);
        context.out << str << std::endl;

        return str.empty();
    }

    using namespace std;

    int main()
    {
        Strings s;

        s.uppercase(cout);
    }
```

Note that only the class `Strings` needed to be modified. The `Wrap1` template could be used to create both the plain, `void` returning function object and the unary predicate.

A final note: sometimes no context is required at all, but the approach taken with the `Wrap1` template class may be considered useful. In those cases, either a dummy context may be defined, or a alternate wrapper class not using a context may be defined. Personally, I've done the latter.

20.7.4 The configurable, two argument function object template

Having constructed the unary template wrapper, the construction of the binary template wrapper should offer no surprises. The function object's `operator()()` is now called with two, rather than one argument. Coining the classname `Wrap2`, it's implementation is almost identical to `Wrap1`'s implementation. It's full implementation consists of slightly over 1100 lines, due to the various combinations of `const` and pointer or reference parameters and can be found in the `Bobcat library`[3]. . An excerpt from that class, showing the use of `const *` parameters, is:

```
template <typename Type1, typename Type2,
          typename Context, typename ReturnType = void>
class Wrap2c
{
    union
    {
        Context *d_context;
        Context const *d_contextconst;
    };
    union
    {
        // from right to left the following is varied:
        //          references, pointers,
        //          const references, const pointers
```

[3]http://bobcat.sourceforge.net

```
        //  This makes for 4 variations on parameter 3,
        //      x 4 variations on parameter 2
        //      x 4 variations on parameter 1 = 64 variations

        // 3 sets of 16 variations not shown
        // set 4: 16  variations for Type1 const *:
            // 12 variations not shown.

        ReturnType (*d_cptr2ref)(Type1 const *, Type2 const *,
                                                Context &);
        ReturnType (*d_cptr2ptr)(Type1 const *, Type2 const *,
                                                Context *);
        ReturnType (*d_cptr2cref)(Type1 const *, Type2 const *,
                                                Context const &);
        ReturnType (*d_cptr3)(Type1 const *, Type2 const *,
                                                Context const *);
    };

    public:
        typedef Type1       first_argument_type;
        typedef Type2       second_argument_type;
        typedef ReturnType  result_type;

        // Type1 may be &, const &, * and const *
        // Type2 may be &, const &, * and const *
        // Context may be &, const &. * and const *
        // This allows for 64 combinations

    // Three blocks of 16 constructors not shown

    // Fourth block of 16 constructors: Type1 const *
        // Type 1: const *, Type2: &            - not shown
        // Type 1: const *, Type2: *            - not shown
        // Type 1: const *, Type2: const &      - not shown

        // Type 1: const *, Type2: const *
        Wrap2c(ReturnType (*fun)(Type1 const *, Type2 const *, Context &),
               Context &context);
        Wrap2c(ReturnType (*fun)(Type1 const *, Type2 const *, Context *),
               Context *context);
        Wrap2c(ReturnType (*fun)(Type1 const *, Type2 const *,
                                 Context const &), Context const &context);
        Wrap2c(ReturnType (*fun)(Type1 const *, Type2 const *,
                                 Context const *), Context const *context);

    // Member functions: 16, for ref, ptr, const ref, const ptr,
    //                         and two parameters per function
    // Type 1: ref          - not shown
    // Type 1: ptr          - not shown
    // Type 2: const ref    - not shown

    // Type 2: const ptr
        // ...
        ReturnType operator()(Type1 const *param1, Type2 const *param2) const;
```

```
};

// Fourth block of 16 constructors: Type1 const *

    // Type 1: const *, Type2: const *

    template<typename Type1, typename Type2, typename Context,
             typename ReturnType>
    Wrap2c<Type1, Type2, Context, ReturnType>::
    Wrap2c(ReturnType (*fun)(Type1 const *, Type2 const *, Context &),
           Context &context)
    :
        d_context(&context),
        d_cptr2ref(fun)
    {}

    template<typename Type1, typename Type2, typename Context,
             typename ReturnType>
    Wrap2c<Type1, Type2, Context, ReturnType>::
    Wrap2c(ReturnType (*fun)(Type1 const *, Type2 const *, Context *),
           Context *context)
    :
        d_context(context),
        d_cptr2ptr(fun)
    {}

    template<typename Type1, typename Type2, typename Context,
             typename ReturnType>
    Wrap2c<Type1, Type2, Context, ReturnType>::
    Wrap2c(ReturnType (*fun)(Type1 const *, Type2 const *, Context const &),
           Context const &context)
    :
        d_contextconst(&context),
        d_cptr2cref(fun)
    {}

    template<typename Type1, typename Type2, typename Context,
             typename ReturnType>
    Wrap2c<Type1, Type2, Context, ReturnType>::
    Wrap2c(ReturnType (*fun)(Type1 const *, Type2 const *, Context const *),
           Context const *context)
    :
        d_contextconst(context),
        d_cptr3(fun)
    {}

// Member functions: 16, for ref, ptr, const ref, const ptr,
//                       and two parameters per function
// ...

// Type 2: const ptr
    // ...

    template<typename Type1, typename Type2, typename Context,
```

```
            typename ReturnType>
    ReturnType Wrap2c<Type1, Type2, Context, ReturnType>::
    operator()(Type1 const *param1, Type2 const *param2) const
    {
        return (*d_cref2ref)(*param1, *param2, *d_context);
    }
```

As with the unary template wrapper (see section 20.7.3), an additional class may be defined that does not require a local context.

20.8 Using 'bisonc++' and 'flex'

The example discussed in this section digs into the peculiarities of using a parser- and scanner generator generating **C++** sources. Once the input for a program exceeds a certain level of complexity, it's advantageous to use a scanner- and parser-generator to create 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'`[4].

However, scanner- and parser generators are also (and maybe even more commonly, nowadays) available as free software. Both `bison` and `flex` are usually part of software distributions or they can be obtained from `ftp://prep.ai.mit.edu/pub/non-gnu`. `Flex` creates a `C++` class when `%option c++` is specified.

For parser generators the program `bison` is available. Back in the early 90's *Alain Coetmeur* (`coetmeur@icdc.fr`[5]) created a **C++** variant (`bison++`) creating a parser class. Although `bison++` program produces code that can be used in **C++** programs it also shows many characteristics that are more appropriate in a **C** context than in a **C++** context. In January 2005 I rewrote parts of Alain's `bison++` program, resulting in the original version of the program **bisonc++**. Then, in May 2005 a complete rewrite of the `bisonc++` parser gegerator was completed, which is available on the Internet having version numbers 0.98 and beyond. `Bisonc++` can be downloaded from `http://bisoncpp.sourceforge.net/`, where it is available as source archive and as binary (i386) `Debian`[6] binary package (including `bisonc++`'s documentation). `Bisonc++` creates a cleaner parser class setup than `bison++`. In particular, it derives the parser class from a base-class, containing the parser's token- and type-definitions as well as all member functions which should not be (re)defined by the programmer. Most of these members might also be defined directly in the parser class. Because of this approach, the resulting parser class is very small, declaring only members that are actually defined by the programmer (as well as some other members, generated by `bisonc++` itself, implementing the parser's `parse()` member). Actually, `parse()` is initially the *only* public member of `bisonc++`'s generated parser class. Remaining members are private. The only member which is *not* implemented by default is `lex()`, producing the next lexical token. When the directive `%scanner` (see section 20.8.2.1) is used, `bisonc++` will generate a standard implementation for this member; otherwise it must be implemented by the programmer.

In this section of the Annotations we will focus on `bisonc++` as our *parser generator*.

Using `flex` and `bisonc++` `class`-based scanners and parsers can be generated. The advantage of this approach is that the interface to the scanner and the parser tends to become cleaner than

[4] http://www.oreilly.com/catalog/lex
[5] mailto:coetmeur@icdc.fr
[6] http://www.debian.org

without using the `class` interface. Furthermore, classes allow us to get rid of most if not all global variables, making it easy to use multiple parsers in one program.

Below two examples are elaborated. The first example only uses `flex`. The scanner it generates monitors 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 `flex` and `bisonc++` to generate a scanner and a parser transforming standard arithmetic expressions to their postfix notations, commonly used in code generated by compilers and in `HP`-calculators. In the second example the emphasis is mainly on `bisonc++` and on composing a scanner object inside a generated parser.

20.8.1 Using 'flex' to create a scanner

The lexical scanner developed in this section is used to monitor the production of a file from several subfiles. The setup is as follows: the input-language knows of an `#include` directive, followed by a text string specifying the file (path) which should be included at the location of the `#include`.

In order to avoid complexities irrelevant to the current example, the format of the `#include` statement is restricted to the form `#include <filepath>`. The file specified between the pointed brackets should be available at the location indicated by `filepath`. If the file is not available, the program terminates after issuing an error message.

The program is started with one or two filename arguments. If the program is started with just one filename argument, the output is written to the standard output stream `cout`. Otherwise, the output is written to the stream whose name is given as the program's second argument.

The program defines a maximum nesting depth. Once this maximum is exceeded, the program terminates after issuing an error message. In that case, the filename stack indicating where which file was included is printed.

One additional feature is that (standard **C++**) comment-lines are ignored. So, `include` directives in comment-lines are ignored too.

The program is created along the following steps:

- First, the file `lexer` is constructed, containing the input-language specifications.
- From the specifications in `lexer` the requirements for the `class Scanner` evolve. The `Scanner` class is a wrapper around the class `yyFlexLexer` generated by `flex`. The requirements result in the interface specification for the class `Scanner`.
- Next, `main()` is constructed. A `Scanner` object is created inspecting 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 various classes are constructed.
- Finally, the program is compiled and linked.

20.8.1.1 The derived class 'Scanner'

The code associated with the regular expression rules is located inside the class `yyFlexLexer`. However, we would of course want to use the derived class's members in this code. This causes a little problem: how does a base-class member know about members of classes derived from it?

Fortunately, inheritance helps us to realize this. In the specification of the class `yyFlexLexer()`, we notice that the function `yylex()` is a *virtual* function. The header file `FlexLexer.h` declares the `virtual` member `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 );
};
```

As this function is a virtual function it can be overridden in a *derived* class. In that case the overridden function will be called from its base class (i.e., `yyFlexLexer`) code. Since the derived class's `yylex()` is called, it will now have access to the members of the derived class, and also to the public and protected members of its base class.

By default, the context in which the generated scanner is placed is the function `yyFlexLexer::yylex()`. This context changes if we use a derived class, e.g., `Scanner`. To derive `Scanner` from `yyFlexLexer`, generated by `flex`, do as follows:

- The function `yylex()` must be declared in the derived class `Scanner`.
- *Options* (see below) are used to inform `flex` about the derived class's name.

Looking at the regular expressions themselves, notice that we need rules to recognize comment, `#include` directives, and all remaining characters. This is all fairly standard practice. When an `#include` directive is detected, the directive is parsed by the scanner. This too is common practice. Here is what our lexical scanner will do:

- As usual, preprocessor directives are not analyzed by a parser, but by the lexical scanner;
- The scanner uses a mini scanner to extract the filename from the directive, throwing a `Scanner::Error` value (`invalidInclude`) if this fails;
- If the filename could be extracted, it is stored in `nextSource`;
- When the `#include` directive has been processed, `pushSource()` is called to perform the switch to another file;
- When the end of the file (`EOF`) is reached, the derived class's member function `popSource()` is called, popping the previously pushed file and returning `true`;
- Once the file-stack is empty, `popSource()` returns `false`, resulting in calling `yyterminate()`, terminating the scanner.

The lexical scanner specification file is organized similarly as the one used for `flex` in **C** contexts. However, for **C++** contexts, `flex` may create a class (`yyFlexLexer`) from which another class (e.g., `Scanner`) can be derived. The flex specification file itself has three sections:

- The lexer specification file's first section is a **C++** *preamble*, containing code which can be used in the code defining the actions to be performed once a regular expression is matched. In the current setup, where each class has its own *internal header file*, the internal header file includes the file `scanner.h`, in turn including `FlexLexer.h`, which is part of the `flex` distribution.

  However, due to the complex setup of this latter file, it should not be read again by the code generated by `flex`. So, we now have the following situation:

  - First we look at the lexer specification file. It contains a preamble including `scanner.ih`, since this declares, via `scanner.h` the class `Scanner`, so that we're able to call `Scanner`'s members from the code associated with the regular expressions defined in the lexer specification file.
  - In `scanner.h`, defining `class Scanner`, the header file `FlexLexer.h`, declaring `Scanner`'s base class, *must* have been read by the compiler before the class `Scanner` itself is defined.
  - Code generated by `flex` already includes `FlexLexer.h`, and as mentioned, `FlexLexer.h` may not be read again. However, `flex` will also insert the specification file's preamble into the code it generates.
  - Since this preamble includes `scanner.ih`, and so `scanner.h`, and so `FlexLexer.h`, we now *do* include `FlexLexer.h` twice in code generated by `flex`. This must be prevented.

  To prevent multiple inclusions of `FlexLexer.h` the following is suggested:

  - Although `scanner.ih` includes `scanner.h`, `scanner.h` itself is modified such that it includes `FlexLexer.h`, *unless* the **C** preprocesser variable `_SKIP_FLEXLEXER_` is defined.
  - In `flex`' specification file `_SKIP_FLEXLEXER_` is defined just prior to including `scanner.ih`.

  Using this scheme, code generated by `flex` will now re-include `FlexLexer.h`. At the same time, compiling `Scanner`'s members proceeds independently of the lexer specification file's preamble, so here `FlexLexer.h` is properly included too. Here is the specification files' preamble:

```
%{
    #define _SKIP_YYFLEXLEXER_
    #include "scanner.ih"
%}
```

- The specification file's second section is a `flex` *symbol area*, used to define symbols, like a mini scanner, or *options*. The following options are suggested:

  - `%option 8bit`: this allows the generated lexical scanner to read 8-bit characters (rather than 7-bit, which is the default).
  - `%option c++`: this results in `flex` generating **C++** code.
  - `%option debug`: this will include *debugging* code into the code generated by `flex`. Calling the member function `set_debug(true)` will activate this debugging code run-time. When activated, information about which rules are matched is written to the standard error stream. To suppress the execution of debug code the member function `set_debug(false)` may be called.
  - `%option noyywrap`: when the scanner reaches the end of file, it will (by default) call a function `yywrap()` which may perform the switch to another file to be processed. Since there exist alternatives which render this function superfluous (see below), it is suggested to specify this option as well.

- `%option outfile="yylex.cc"`: this defines `yylex.cc` as the name of the generated **C++** source file.
- `%option warn`: this option is strongly suggested by the `flex` documentation, so it's mentioned here as well. See `flex`' documentation for details.
- `%option yyclass="Scanner"`: this defines `Scanner` as the name of the class derived from `yyFlexLexer`.
- `%option yylineno`: this option causes the lexical scanner to keep track of the line numbers of the files it is scanning. When processing nested files, the variable `yylineno` is not automatically reset to the last line number of a file, when returning to a partially processed file. In those cases, `yylineno` will explicitly have to be reset to a former value. If specified, the current line number is returned by the public member `lineno()`, returning an `int`.

Here is the specification files' symbol area:

```
%option yyclass="Scanner" outfile="yylex.cc" c++ 8bit warn noyywrap yylineno
%option debug

%x      comment
%x      include

eolnComment     "//".*
anyChar         .|\n
```

- The specification file's third section is a *rules section*, in which the regular expressions and their associated actions are defined. In the example developed here, the lexer should copy information from the `istream *yyin` to the `ostream *yyout`. For this the predefined macro `ECHO` can be used. Here is the specification files' symbol area:

```
%%
    /*
        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>]+       d_nextSource = yytext;
<include>">"[ \t]*\n    {
                            BEGIN INITIAL;
                            pushSource(YY_CURRENT_BUFFER, YY_BUF_SIZE);
                        }
<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(YY_CURRENT_BUFFER))
                                yyterminate();
                        }
%%
```

Since the derived class’s members may now 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), most processing can be left to the derived class’s member functions. This results in a very clean setup of the lexer specification file, requiring no or hardly any code in the *preamble*.

20.8.1.2 Implementing ‘Scanner’

The `class Scanner` is derived from the class `yyFlexLexer`, generated by `flex`. The derived class has access to data controlled by the lexical scanner. In particular, it has access to the following data members:

- `char *yytext`, containing the text matched by a regular expression. Clients may access this information using the scanner’s `YYText()` member;
- `int yyleng`, the length of the text in `yytext`. Clients may access this value using the scanner’s `YYLeng()` member;
- `int yylineno`: the current line number. This variable is only maintained if `%option yylineno` is specified. Clients may access this value using the scanner’s `lineno()` member.

Other members are available as well, but are used less often. Details can be found in `FlexLexer.h`.

Objects of the `class Scanner` perform two tasks:

- They push file information about the current file to a file stack;
- They pop the last-pushed information from the stack once `EOF` is detected in a file.

Several member functions are used to accomplish these tasks. As they are auxiliary to the scanner, they are private members. In practice, develop these private members once the need for them arises. Note that, apart from the private member functions, several private data members are defined as well. Let’s have a closer look at the implementation of the class `Scanner`:

- First, we have a look at the class’s initial section, showing the conditional inclusion of `FlexLexer.h`, its `class` opening, and its private data. Its `public` section starts off by defining the `enum Error` defining various symbolic constants for errors that may be detected:

  ```
  #if ! defined(_SKIP_YYFLEXLEXER_)
  #include <FlexLexer.h>
  #endif

  class Scanner: public yyFlexLexer
  {
  ```

```
    std::stack<yy_buffer_state *>   d_state;
    std::vector<std::string>        d_fileName;
    std::string                     d_nextSource;

    static size_t const           s_maxDepth = 10;

    public:
        enum Error
        {
            invalidInclude,
            circularInclusion,
            nestingTooDeep,
            cantRead,
        };
```

- As they are objects, the class's data members are initialized automatically by `Scanner`'s constructor. It activates the initial input (and output) file and pushes the name of the initial input file. Here is its implementation:

```
#include "scanner.ih"

Scanner::Scanner(istream *yyin, string const &initialName)
{
    switch_streams(yyin, yyout);
    d_fileName.push_back(initialName);
}
```

- The scanning process proceeds as follows: once the scanner extracts a filename from an `#include` directive, a switch to another file is performed by `pushSource()`. If the filename could not be extracted, the scanner throws an `invalidInclude` exception value. The `pushSource()` member and the matching function `popSource()` handle file switching. Switching to another file proceeds as follows:
  - First, the current depth of the `include`-nesting is inspected. If `s_maxDepth` is reached, the stack is considered full, and the scanner throws a `nestingTooDeep` exception.
  - Next, `throwOnCircularInclusion()` is called to avoid circular inclusions when switching to new files. This function throws an exception if a filename is included twice, using a simple literal name check. Here is its implementation:

```
#include "scanner.ih"

void Scanner::throwOnCircularInclusion()
{
    vector<string>::iterator
        it = find(d_fileName.begin(), d_fileName.end(), d_nextSource);

    if (it != d_fileName.end())
        throw circularInclusion;
}
```

  - 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()`.

Here is pushSource()'s implementation:

```
#include "scanner.ih"

void Scanner::pushSource(yy_buffer_state *current, size_t size)
{
    if (d_state.size() == s_maxDepth)
        throw nestingTooDeep;

    throwOnCircularInclusion();
    d_fileName.push_back(d_nextSource);

    ifstream *newStream = new ifstream(d_nextSource.c_str());

    if (!*newStream)
        throw cantRead;

    d_state.push(current);
    yy_switch_to_buffer(yy_create_buffer(newStream, size));
}
```

- The class yyFlexLexer provides a series of member functions that can be used to switch files. 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 currently read file. This buffer is pushed on the d_state stack when an #include is encountered. Then yy_buffer_state's contents are replaced by the buffer created for the file to be processed next. Note that in the flex specification file the function pushSource() is called as

```
    pushSource(YY_CURRENT_BUFFER, YY_BUF_SIZE);
```

  YY_CURRENT_BUFFER and YY_BUF_SIZE are macros that are *only* available in the rules section of the lexer specification file, so they must be passed as arguments to pushSource(). Currently it is *not* possible to use these macros in the Scanner class's member functions directly.

- 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 pushSource(), whereas popSource() should reinstate a former value of yylineno by popping a previously pushed value from the stack. Scanner's current implementation maintains a simple stack of yy_buffer_state pointers. Changing that into a stack of pair<yy_buffer_state *, size_t> elements would allow us to save (and restore) line numbers as well. This modification is left as an exercise to the reader.

- The member function popSource() is called to pop the previously pushed buffer from the stack, allowing the scanner to continue its scan just beyond the just processed #include directive. The member popSource() first inspects the size of the d_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 pushSource(). Furthermore, the filename that was last entered in the d_fileName vector is removed. Having done all this, the function returns true:

```
#include "scanner.ih"

bool Scanner::popSource(yy_buffer_state *current)
```

```
    {
        if (d_state.empty())
            return false;

        yy_delete_buffer(current);
        yy_switch_to_buffer(d_state.top());
        d_state.pop();
        d_fileName.pop_back();

        return true;
    }
```

- Two service members are offered: `stackTrace()` dumps the names of the currently pushed files to the standard error stream. It may be called by exception catchers. Here is its implementation:

```
    #include "scanner.ih"

    void Scanner::stackTrace()
    {
        for (size_t idx = 0; idx < d_fileName.size() - 1; ++idx)
            cerr << idx << ": " << d_fileName[idx]  << " included " <<
                                    d_fileName[idx + 1]  << endl;
    }
```

- `lastFile()` returns the name of the currently processed file. It may be implemented inline:

```
inline std::string const &Scanner::lastFile()
{
    return d_fileName.back();
}
```

- The lexical scanner itself is defined in `Scanner::yylex()`. Therefore, `int yylex()` must be declared by the class `Scanner`, as it overrides `FlexLexer`'s virtual member `yylex()`.

20.8.1.3 Using a 'Scanner' object

The program using our `Scanner` is very simple. It expects a filename indicating where to start the scanning process. Initially the number of arguments is checked. If at least one argument was given, then an `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 initial input file as its arguments. Then the `Scanner` object's `yylex()` member is called. The scanner object throws `Scanner::Error` exceptions if it fails to perform its tasks properly. These exceptions are caught near `main()`'s end. Here is the program's source:

```
#include "lexer.h"
using namespace std;

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
        {
            return 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;
        }
        return 0;
    }
```

20.8.1.4 Building the 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 following command can be given:

  ```
  flex lexer
  ```

- Next, all sources are compiled and linked. In situations where the default `yywrap()` function is used, the `libfl.a` library should be linked against the final program. Normally, that's not required, and the program can be constructed as, e.g.:

  ```
  g++ -o lexer *.cc
  ```

For the purpose of debugging a lexical scanner, the matched rules and the returned tokens provide useful information. When the `%option debug` was specified, debugging code will be included in the generated scanner. To obtain debugging info, this code must also be activated. Assuming the scanner object is called `scanner`, the statement

```
scanner.set_debug(true);
```

will produce debugging info to the standard error stream.

20.8.2 Using both 'bisonc++' and 'flex'

When an input language exceeds a certain level of complexity, a *parser* is often used to control the complexity of the input language. In this case, a *parser generator* can be used to generate the code verifying the input's grammatical correctness. The lexical scanner (preferably composed into the parser) provides chunks of the input, called *tokens*. The parser then processes the series of tokens generated by its lexical scanner.

Starting point when developing programs that use both parsers and scanners is the grammar. The grammar defines 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 performed by the parser and by the lexer are not normally specified literally in the grammatical rules or lexical regular expressions, but should be implemented in *member functions*, called from within the parser's rules or which are 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 concentrate on the parser. The parser can be generated from a grammar specification, processed by the program `bisonc++`. The grammar specification required for `bisonc++` is similar to the specifications required for `bison` (and an existing program `bison++`, written in the early nineties by the Frenchman *Alain Coetmeur*), but `bisonc++` generates a **C++** which more closely follows present-day standards than `bison++`, which still shows many **C**-like features.

In this section a program is developed converting *infix expressions*, in which binary operators are written between their operands, to *postfix expressions*, in which binary operators are written behind their operands. Furthermore, the unary operator `-` will be converted from its prefix notation to a postfix form. The unary `+` operator is ignored as it requires no further actions. In essence our little calculator is a micro compiler, transforming numerical expressions into assembly-like instructions.

Our calculator will recognize a very basic set of operators: multiplication, addition, parentheses, and the unary minus. We'll distinguish real numbers from integers, to illustrate a subtlety in bison-like grammar specifications. That's all. The purpose of this section is, after all, to illustrate the construction of a **C++** program that uses both a parser and a lexical scanner, rather than to construct a full-fledged calculator.

In the coming sections we'll develop the grammar specification for `bisonc++`. Then, the regular expressions for the scanner are specified according to `flex`' requirements. Finally the program is constructed.

20.8.2.1 The 'bisonc++' specification file

The grammar specification file required by `bisonc++` is comparable to the specification file required by `bison`. Differences are related to the class nature of the resulting parser. Our calculator will distinguish real numbers from integers, and will support a basic set of arithmetic operators.

`Bisonc++` should be used as follows:

- As usual, a grammar must be defined. With `bisonc++` this is no different, and `bisonc++` grammar definitions are for all practical purposes identical to `bison`'s grammar definitions.
- Having specified the grammar and (usually) some declarations `bisonc++` is able to generate files defining the parser class and the implementation of the member function `parse()`.
- All class members (except those that are required for the proper functioning of the member `parse()`) must be implemented by the programmer. Of course, they should also be declared

in the parser class's header. At the very least the member `lex()` must be implemented. This member is called by `parse()` to obtain the next available token. However, `bisonc++` offers a facility providing a standard implementation of the function **lex()**. The member function `error(char const *msg)` is given a simple default implementation which may be modified by the programmer. The member function `error()` is called when `parse()` detects (syntactical) errors.

- The parser can now be used in a program. A very simple example would be:

```
int main()
{
    Parser parser;
    return parser.parse();
}
```

The `bisonc++` specification file consists of two sections:

- The *declaration section*. In this section bison's tokens, and the priority rules for the operators are declared. However, `bisonc++` also supports several new declarations. These new declarations are important and are discussed below.
- The *rules section*. The grammatical rules define the grammar. This section is identical to the one required by `bison`, albeit that some members that were available in `bison` and `bison++` are considered obsolete in `bisonc++`, while other members can now be used in a wider context. For example, **ACCEPT()** and **ABORT()** can be called from any member called from the parser's action blocks to terminate the parsing process.

Readers familiar with `bison` should note that there is no *header section* anymore. Header sections are used by bison to provide for the necessary declarations allowing the compiler to compile the **C** function generated by `bison`. In **C++** declarations are part of or already used by class definitions. Therefore, a parser generator generating a **C++** class and some of its member functions does not require a header section anymore.

The declaration section The declaration section contains several declarations, among which all tokens used in the grammar and the priority rules of the mathematical operators. Moreover, several new and important specifications can be used here. Those that are relevant to our current example and only available in `bisonc++` are discussed here. The reader is referred to `bisonc++`'s *man-page* for a full description.

- **%baseclass-header** `header`
  Defines the pathname of the file to contain (or containing) the parser's base class. Defaults to the name of the parser class plus the suffix `base.h`.
- **%baseclass-preinclude** `header`
  Use `header` as the pathname to the file pre-included in the parser's base-class header. This declaration is useful in situations where the base class header file refers to types which might not yet be known. E.g., with **%union** a **std::string** * field might be used. Since the class **std::string** might not yet be known to the compiler once it processes the base class header file we need a way to inform the compiler about these classes and types. The suggested procedure is to use a pre-include header file declaring the required types. By default `header` will be surrounded by double quotes (using, e.g., `#include "header"`). When the argument is surrounded by angle brackets `#include <header>` will be included. In the latter case, quotes might be required to escape interpretation by the shell (e.g., using `-H '<header>'`).

- **%class-header** `header`
  Defines the pathname of the file to contain (or containing) the parser class. Defaults to the name of the parser class plus the suffix `.h`

- **%class-name** `parser-class-name`
  Declares the class name of this parser. This declaration replaces the **%name** declaration previously used by `bison++`. It defines the name of the **C++** class that will be generated. Contrary to `bison++`'s **%name** declaration, **%class-name** may appear anywhere in the first section of the grammar specification file. It may be defined only once. If no **%class-name** is specified the default class name `Parser` will be used.

- **%debug**
  Provide **parse()** and its support functions with debugging code, showing the actual parsing process on the standard output stream. When included, the debugging output is active by default, but its activity may be controlled using the **setDebug(bool on-off)** member. Note that no `#ifdef DEBUG` macros are used anymore. By rerunning `bisonc++` without the **–debug** option an equivalent parser is generated not containing the debugging code.

- **%filenames** `header`
  Defines the generic name of all generated files, unless overridden by specific names. By default the generated files use the class-name as the generic file name.

- **%implementation-header** `header`
  Defines the pathname of the file to contain (or containing) the implementation header. Defaults to the name of the generated parser class plus the suffix `.ih`. The implementation header should contain all directives and declarations *only* used by the implementations of the parser's member functions. It is the only header file that is included by the source file containing `parse()`'s implementation. It is suggested that user defined implementations of other class members use the same convention, thus concentrating all directives and declarations that are required for the compilation of other source files belonging to the parser class in one header file.

- **%parsefun-source** `source`
  Defines the pathname of the file containing the parser member **parse()**. Defaults to `parse.cc`.

- **%scanner** `header`
  Use `header` as the pathname to the file pre-included in the parser's class header. This file should define a class **Scanner**, offering a member `int yylex()` producing the next token from the input stream to be analyzed by the parser generated by `bisonc++`. When this option is used the parser's member `int lex()` will be predefined as (assuming the parser class name is `Parser`):

  ```
  inline int Parser::lex()
  {
      return d_scanner.yylex();
  }
  ```

  and an object `Scanner d_scanner` will be composed into the parser. The `d_scanner` object will be constructed using its default constructor. If another constructor is required, the parser class may be provided with an appropriate (overloaded) parser constructor after having constructed the default parser class header file using `bisonc++`. By default `header` will be surrounded by double quotes (using, e.g., `#include "header"`). When the argument is surrounded by angle brackets `#include <header>` will be included.

- **%stype typename**
  The type of the semantic value of tokens. The specification `typename` should be the name of an unstructured type (e.g., `size_t`). By default it is `int`. See **YYSTYPE** in `bison`. It should

not be used if a **%union** specification is used. Within the parser class, this type may be used as `STYPE`.

- **%union** `union-definition`
  Acts identically to the `bison` declaration. As with `bison` this generates a union for the parser's semantic type. The union type is named `STYPE`. If no **%union** is declared, a simple stack-type may be defined using the **%stype** declaration. If no **%stype** declaration is used, the default stacktype (`int`) is used.

An example of a `%union` declaration is:

```
%union
{
    int     i;
    double  d;
};
```

A union cannot contain objects as its fields, as constructors cannot be called when a union is created. This means that a `string` cannot be a member of the union. A `string *`, however, *is* a possible union member. By the way: the lexical scanner does not have to know about such a union. The scanner can simply pass its scanned text to the parser through its `YYText()` member function. For example, using a statement like

```
$$.i = A2x(scanner.YYText());
```

matched text may be converted to a value of an appropriate type.

Tokens and non-terminals can be associated with union fields. This is strongly advised, as it prevents type mismatches, since the compiler will be able to check for type correctness. At the same time, the bison specific variabels `$$`, `$1`, `$2`, etc. may be used, rather than the full field specification (like `$$.i`). A non-terminal or a token may be associated with a union field using the `<fieldname>` specification. E.g.,

```
%token <i> INT          // token association (deprecated, see below)
       <d> DOUBLE
%type  <i> intExpr      // non-terminal association
```

In the example developed here, note that both the tokens and the non-terminals can be associated with a field of the union. However, as noted before, 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*, knowing what the input is all about, may then convert strings like `"123"` to an integer value. Consequently, the association of a union field and a token is discouraged. In the upcoming description of the rules of the grammar this will be illustrated further.

In the `%union` discussion the `%token` and `%type` specifications should be noted. They are used to specify the tokens (terminal symbols) that can be returned by the lexical scanner, and to specify 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 `bisonc++` practice, it isn't further elaborated here. The documentation provided with `bisonc++`'s distribution should be consulted for further reference.

Here is the specification of the calculator's declaration section:

```
%filenames parser
%scanner ../scanner/scanner.h
%lines

%union {
    int i;
    double d;
};

%token  INT
        DOUBLE

%type   <i> intExpr
%type   <d> doubleExpr

%left   '+'
%left   '*'
%right  UnaryMinus
```

In the declaration section `%type` specifiers are used, associating the `intExpr` rule's value (see the next section) to the `i`-field of the semantic-value union, and associating `doubleExpr`'s value to the `d`-field. At first sight this may look 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 still have had to specify somewhere in the returned semantic values what field to use: less rules, but more complex and error-prone code.

The grammar rules The rules and actions of the grammar are specified as usual. The grammar for our little calculator is given below. There are quite a few rules, but they illustrate various features offered by `bisonc++`. In particular, note that no action block requires more than a single line of code. This keeps the organization of the grammar relatively simple, and therefore enhances its readability and understandability. Even the rule defining the parser's proper termination (the empty line in the `line` rule) uses a single member function call `done()`. The implementation of that function is simple, but interesting in that it calls **Parser::ACCEPT()**, showing that the **ACCEPT()** member can be called indirectly from a production rule's action block. Here are the grammar's production rules:

```
    lines:
        lines
        line
    |
        line
    ;

    line:
        intExpr
        '\n'
        {
            display($1);
        }
    |
        doubleExpr
```

```
        '\n'
        {
            display($1);
        }
    |
        '\n'
        {
            done();
        }
    |
        error
        '\n'
        {
            reset();
        }
    ;

    intExpr:
        intExpr '*' intExpr
        {
            $$ = exec('*', $1, $3);
        }
    |
        intExpr '+' intExpr
        {
            $$ = exec('+', $1, $3);
        }
    |
        '(' intExpr ')'
        {

            $$ = $2;
        }
    |
        '-' intExpr         %prec UnaryMinus
        {
            $$ = neg($2);
        }
    |
        INT
        {
            $$ = convert<int>();
        }
    ;

    doubleExpr:
        doubleExpr '*' doubleExpr
        {
            $$ = exec('*', $1, $3);
        }
    |
        doubleExpr '*' intExpr
        {
            $$ = exec('*', $1, d($3));
```

```
        }
    |
        intExpr '*' doubleExpr
        {
            $$ = exec('*', d($1), $3);
        }
    |
        doubleExpr '+' doubleExpr
        {
            $$ = exec('+', $1, $3);
        }
    |
        doubleExpr '+' intExpr
        {
            $$ = exec('+', $1, d($3));
        }
    |
        intExpr '+' doubleExpr
        {
            $$ = exec('+', d($1), $3);
        }
    |
        '(' doubleExpr ')'
        {
            $$ = $2;
        }
    |
        '-' doubleExpr         %prec UnaryMinus
        {

            $$ = neg($2);
        }
    |
        DOUBLE
        {
            $$ = convert<double>();
        }
    ;
```

The above grammar is used to implement a simple calculator in which integer and real values can be negated, added, and multiplied, and in which standard priority rules can be circumvented using parentheses. The grammar shows the use of typed nonterminal symbols: `doubleExpr` is linked to real (double) values, `intExpr` is linked to integer values. Precedence and type association is defined in the parser's definition section.

The Parser's header file Various functions called from the grammar are defined as template functions. `Bisonc++` generates various files, among which the file defining the parser's class. Functions called from the production rule's action blocks are usually member functions of the parser, and these member functions must be declared and defined. Once `bisonc++` has generated the header file defining the parser's class it will not automatically rewrite that file, allowing the programmer to add new members to the parser class. Here is the `parser.h` file as used for our little calculator:

```
#ifndef Parser_h_included
```

```
#define Parser_h_included

#include <iostream>
#include <sstream>
#include <bobcat/a2x>

#include "parserbase.h"
#include "../scanner/scanner.h"


#undef Parser
class Parser: public ParserBase
{
    std::ostringstream d_rpn;
    // $insert scannerobject
    Scanner d_scanner;

    public:
        int parse();

    private:
        template <typename Type>
            Type exec(char c, Type left, Type right);
        template <typename Type>
            Type neg(Type op);
        template <typename Type>
            Type convert();

        void display(int x);
        void display(double x);
        void done() const;
        void reset();
        void error(char const *msg);
        int lex();
        void print();

        static double d(int i);

    // support functions for parse():

        void executeAction(int d_production);
        void errorRecovery();
        int lookup();
        void nextToken();
};


inline double Parser::d(int i)
{
    return i;
}

template <typename Type>
Type Parser::exec(char c, Type left, Type right)
```

```
{
    d_rpn << " " << c << " ";
    return c == '*' ? left * right : left + right;
}

template <typename Type>
Type Parser::neg(Type op)
{
    d_rpn << " n ";
    return -op;
}

template <typename Type>
Type Parser::convert()
{
    Type ret = FBB::A2x(d_scanner.YYText());
    d_rpn << " " << ret << " ";
    return ret;
}

inline void Parser::error(char const *msg)
{
    std::cerr << msg << std::endl;
}

inline int Parser::lex()
{
    return d_scanner.yylex();
}

inline void Parser::print()
{}

#endif
```

20.8.2.2 The 'flex' specification file

The flex-specification file 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:

```
%{
    #define _SKIP_YYFLEXLEXER_
    #include "scanner.ih"

    #include "../parser/parserbase.h"
%}

%option yyclass="Scanner" outfile="yylex.cc" c++ 8bit warn noyywrap
%option debug

%%
```

```
[ \t]                   ;
[0-9]+                  return Parser::INT;

"."[0-9]*               |
[0-9]+("."[0-9]*)?      return Parser::DOUBLE;

.|\n                    return *yytext;

%%
```

20.8.2.3 Generating code

The code is generated in the same way as with `bison` and `flex`. In order to have `bisonc++` generate the files `parser.cc` and `parser.h`, issue the command:

```
bisonc++ -V grammar
```

The option `-V` will generate the file `parser.output` showing information about the internal structure of the provided grammar, among which its states. It is useful for debugging purposes, and can be left out of the command if no debugging is required. `Bisonc++` may detect conflicts (shift-reduce conflicts and/or reduce-reduce conflicts) in the provided grammar. These conflicts may be resolved explicitly, using disambiguation rules or they are ‘resolved’ by default. A shift-reduce conflict is resolved by shifting, i.e., the next token is consumed. A reduce-reduce conflict is resolved by using the first of two competing production rules. `Bisonc++` uses identical conflict resolution procedures as `bison` and `bison++`.

Once a parser class and parsing member function has been constructed `flex` may be used to create a lexical scanner (in, e.g., the file `yylex.cc`) using the command

```
flex -I lexer
```

On Unix systems, linking and compiling the generated sources and the source for the main program (given below) is then realized by a command comparable to:

```
g++ -o calc -Wall *.cc -s
```

Finally, here is a source file in which the `main()` function and the parser object is defined. The parser features the lexical scanner as one of its data members:

```
#include "parser/parser.h"

using namespace std;

int main()
{
    Parser parser;

    cout << "Enter (nested) expressions containing ints, doubles, *, + and "
            "unary -\n"
            "operators. Enter an empty line to stop.\n";

    return parser.parse();
}
```

Bisonc++ can be downloaded from http://bisoncpp.sourceforge.net/. It requires the bobcat library, which can be downloaded from http://bobcat.sourceforge.net/.

Index