# C++ Annotations Version 4.4.2

Frank B. Brokken
(and Karel Kubat until version 4.0.0)
Computing Center, University of Groningen
Landleven 1,
P.O. Box 800,
9700 AV Groningen
The Netherlands
Published at the University of Groningen
ISBN 90 367 0470 7

1994 - 2000

**Abstract**

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

If you want a hard-copy version of the **C++** annotations: that's available in postscript, and other formats in our ftp-site, in several files starting with `cplusplus`.

# 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 'class' concept: structs having functions. The 'object' concept: variables of a `class`.
- Chapter 6: Allocation and returning unused memory: `new`, `delete`, and the function `set_new_handler()`.
- Chapter 7: More About Operator Overloading.
- Chapter 8: Abstract Containers.
- Chapter 9: Static data and functions: components of a class not bound to objects.
- Chapter 10: Classes having pointer members: how to prevent memory leaks and wild pointers.
- Chapter 11: The Standard Template Library, generic algorithms.
- Chapter 12: The **C++** type-safe I/O library.
- Chapter 14: Gaining access to private parts from outside: friend functions and classes.
- Chapter 15: Building classes upon classes: setting up class hierarcies.
- Chapter 16: Polymorphism: changing the behavior of member functions accessed through base class pointers.
- Chapter 13: Exceptions: handling errors where appropriate, rather than where they occur.
- Chapter 17: Templates: using *molds* for code that is type dependent.
- Chapter 18: Several examples of programs written in **C++**.

# Chapter 2

# Introduction

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

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

The version number of this document (currently 4.4.2) is updated when the contents of the document change. The first number is the major number, and will probably not be changed for some time: it indicates a major rewriting. The middle number is increased when new information is added to the document. The last number only indicates small changes; it is increased when, e.g., series of typos are corrected.

This document is published by the Computing Center, University of Groningen, the Netherlands. This document was typeset using the yodl formating system.

 **Direct all correspondence concerning suggestions, additions, improvements or changes in this document to the author:**

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

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

[1]e.g., the Dutch book *De programmeertaal C*, Brokken and Kubat, University of Groningen 1996

### 2.0.1 History of the C++ Annotations

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

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

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

The transition from major version three to major version four was realized by Frank: again new chapters were added, and the source-document was converted from `SGML` to Yodl.

> The **C++** Annotations are *not* freely distributable. Be sure to read the legal notes.
>
> **Reading the annotations beyond this point implies that you are aware of the restrictions that we pose and that you agree with them.**

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

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

This section is modified when the first and second part of the version numbers change. Modifications in versions 1.*.*, 2.*.*, and 3.*.* were not logged.

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

The upgrade from version 4.1.* to 4.2.* was the result of the inclusion of section 3.3.1 about the **bool** data type in chapter 3. The distinction between differences between **C** and **C++** and extensions of the **C** programming languages is (albeit a bit fuzzy) reflected in the introdution 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++**.

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. See section 18.4.1 for details.

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.` the following modifications will be incorporated as well:

- Function-addresses must be obtained using the `&`-operator

- Functions called via pointers to member functions must use the `(this->*pointer)(...)` construction inside member functions of the class in which the pointer to member functions is defined.

Version `4.4.1` again contains new material, and reflects the ANSI/ISO standard (well, I try to have it reflect the ANSI/ISO standard). In version 4.4.1. the following sections and chapters were added:

- Releases over 4.4.1 handle typos and suggestions for improvements I've received by email.
- A section (16.6 about *Run-Time Type Identification*, included as of release 4.4.1.
- A section (16.6.1 about the `dynamic_cast` cast operator. included as of release 4.4.1.
- Minor spellingcorrections were made up to release 4.4.0n.
- A reference to `icmake` and the `C++`-build script was added in release 4.4.0m (see section 2.2.2).
- A section (3.6) about *namespaces*, included as of release 4.4.0i.
- A section (7.6) about the `explicit` keyword, included as of release 4.4.0h.
- A section about constructing *manipulators* (12.3.8), included as of release 4.4.0h.
- A section about overloading the operators `++` and `--` (7.7), included as of release 4.4.0h.
- A rewrite of the chapter about Templates (chapter 17), included as of release 4.4.0h.
- A section (11.2 about `auto_ptr` objects, included as of release 4.4.0g.
- A section (5.8) about nested classes. included as of release 4.4.0f.
- The chapter (12) about `iostreams` was modified, and now contains more information about using manipulators and flags, as well as information about using `strstream` objects. Included as of release 4.4.0e.
- A chapter (11 about the *Standard Template Library* and generic algorithms, included as of release 4.4.0e.
- The full contents of the **C++** Annotations can be inspected in parallel with the annotations themselves when the `html`-format is used. Included as of release 4.4.0d.
- The section (5.4) about `inline` functions was slightly modified, included as of release 4.4.0d.
- A section (7.8 about *function objects*, included as of release 4.4.0d.
- A chapter (8 about the abstract container types, included as of release 4.4.0c.
- A section (2.5.4 about the new syntax used with *casts*, included as of release 4.4.0b.
- A section about the `string` type, included as of release 4.4.0b.
- A section (2.2.2 about compiling **C++** programs, included as of release 4.4.0a.

Version `4.4.0` (and subletters) is a construction version, in which the extras mentioned above are only partially available.

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

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

At some point, I considered version `4.4.1.` to be the final version of the **C++** annotations. However, a section on special I/O functions (section 12.3.5) was added to cover unformated 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.`.

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 an email containing any mistakes you find or corrections you would like to suggest. Subreleases like `4.4.2a` etc. contain bugfixes and typographical corrections.

## 2.2 The history of C++

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

**C++** was originally a 'pre-compiler', similar to the preprocessor of **C**, which converted special constructions in its source code to plain **C**. This code was then compiled by a normal **C** compiler. The 'pre-code', which was read by the **C++** pre-compiler, was usually located in a file with the extension `.cc`, `.C` or `.cpp`. This file would then be converted to a C source file with the extension `.c`, which was compiled and linked.

The nomenclature of **C++** source files remains: the extensions `.cc` and `.cpp` are usually still used. However, the preliminary work of a **C++** pre-compiler is in modern compilers usually included in the actual compilation process. Often compilers will determine the type of a source file by the extension. This holds true for Borland's and Microsoft's **C++** compilers, which assume a **C++** source for an extension `.cpp`. The GNU compiler `gcc`, which is available on many Unix platforms, assumes for **C++** the extension `.cc`.

The fact that **C++** used to be compiled into **C** code is also visible from the fact that **C++** is a superset of **C**: **C++** offers all possibilities of **C**, and more. This makes the transition from **C** to **C++** quite easy. Programmers who are familiar with **C** may start 'programming in **C++**' by using source files with an extension `.cc` or `.cpp` instead of `.c`, and can then just comfortably slide into all the possibilities that **C++** offers. No abrupt change of habits is required.

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

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

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

  ```
  putchar(10);
  ```

  and

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

  are synonyms.

  In contrast, in **C++**, `sizeof('c')` is always 1 (but see also section 3.3.2), while an `int` is still an `int`. As we shall see later (see section 2.5.13), two function calls

  ```
  somefunc(10);
  ```

  and

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

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

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

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

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

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

### 2.2.2 Compiling a C++ program

In order to compile a **C++** program, a **C++** compiler is needed. Considering the free nature of this document, it won't come as a surprise that a *free compiler* is suggested here. The Free Software Foundation provides free **C++** compilers which is, among other places, available in the Debian distribution of Linux. The version that's currenly available on my computer is 2.95.2.

### C++ under MS-Windows

For MS-Windows Cygnus provides the foundation for installing the *Windows port* of the `GNU g++` compiler.

Cygnus is now distributed by RedHat. Currently (october 2000), the cygnus distribution is found at http://sources.redhat.com. When linking to that URL, click on the `cygwin` menu entry and then 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.

### Compiling a C++ source text

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

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

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

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

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

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

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

Using the `icmake` program (to be downloaded from ftp://ftp.icce.rug.nl/icmake-X.YY.tar.gz) a maintenance script can be used to assist in the construction and maintenance of a **C++** program. This script has been tested on **Linux** platforms for several years now. It is described at http://www.icce.rug.nl/docs/programs/Cscript.html

## 2.3 Advantages and pretensions of C++

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

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

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

Concerning the above allegations of **C++**, we think that the following can be concluded. The development of new programs while existing code is reused can also be realized in **C** by, e.g., using function libraries: thus, handy functions can be collected in a library and need not be re-invented with each new program. Still, **C++** offers its specific syntax possibilities for code reuse, apart from function libraries (see chapter 15).

Creating and using new data types is also very well possible in **C**; e.g., by using `struct`s, `typedef`s etc.. From these types other types can be derived, thus leading to `struct`s containing `struct`s and so on.

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

Concerning 'bug proneness' we can say that **C++** indeed uses stricter type checking than **C**. However, most modern **C** compilers implement 'warning levels'; it is then the programmer's choice to 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 `struct`s can be manipulated by dedicated functions. Using such techniques, data hiding can be realized even in **C**; though it needs to be said that **C++** offers special syntactical constructions. In contrast, programmers who prefer to use a global variable `int i` for each counter variable will quite likely not benefit from the concept of data hiding, be it in **C** or **C++**.

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

## 2.4 What is Object-Oriented Programming?

Object-oriented programming propagates a slightly different approach to programming problems than the strategy which is usually used in **C**. The **C**-way is known as a 'procedural approach': a problem is decomposed into subproblems and this process is repeated until the subtasks can be coded. Thus a conglomerate of functions is created, communicating through arguments and

[2]these functions are often present in our **C**-programs, they allocate or abort the program when the memory pool is exhausted

variables, global or local (or `static`).

In contrast, or maybe better: in addition to this, an object-oriented approach identifies the **keywords** in the problem. These keywords are then depicted in a diagram and arrows are drawn between these keywords to define an internal hierarchy. The keywords will be the objects in the implementation and the hierarchy defines the relationship between these objects. The term object is used here to describe a limited, well-defined structure, containing all information about some entity: data types and functions to manipulate the data.

As an example of an object-oriented approach, an illustration follows:

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

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

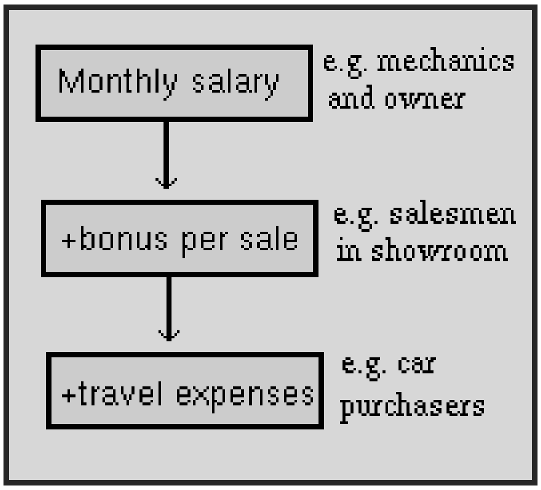
- The owner and the mechanics can be represented as the same type, receiving a given salary per month. The relevant information for such a type would be the monthly amount. In addition this object could contain data as the name, address and social security number.
- Car salesmen who work in the showroom can be represented as the same type as above but with extra functionality: the number of transactions (sales) and the bonus per transaction.

  In the hierarchy of objects we would define the dependency between the first two objects by letting the car salesmen be 'derived' from the owner and mechanics.
- Finally, there are the second-hand car purchasers. These share the functionality of the salesmen except for the travel expenses. The additional functionality would therefore consist of the expenses made and this type would be derived from the salesmen.

The hierarchy of the thus identified objects further illustrated in figure 2.1.

The overall process in the definition of a hierarchy such as the above starts with the description of the most simple type. Subsequently more complex types are derived, while each derivation adds a little functionality. From these derived types, more complex types can be derived *ad infinitum*, until a representation of the entire problem can be made.

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




\

Figure 2.1: Hierarchy of objects in the salary administration.

## 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 End-of-line comment

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

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

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

    return (0);
}
```

The end-of-line comment was already implemented as an extension to **C** in some **C** compilers, such as the Microsoft **C** Compiler V5.

### 2.5.2 NULL-pointers vs. 0-pointers

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

### 2.5.3 Strict type checking

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

The program

```
int main()
{
    printf("Hello World\n");
    return (0);
}
```

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

### 2.5.4 A new syntax for casts

Traditionally, **C** offers the following *cast* construction:

```
(typename)expression
```

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

```
typename(expression)
```

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

- The standard cast to convert one type to another is

```
static_cast<type>(expression)
```

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

[3] When GNU's g++ compiler encounters an unknown function, it assumes that an 'ordinary' C function is meant. It does complain however.

```
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 16): The

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

  is performed run-time to convert, e.g., a pointer to an object of a certain class to a pointer to an object in its so-called *class hierarchy*. At this point in the *Annotations* it is a bit premature to discuss the dynamic_cast, but we will return to this topic in section 16.6.1.

### 2.5.5 The 'static_cast'-operator

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

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

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

```
intVar += doubleVar;
```

Here, the evaluated expression actually is:

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

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

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

### 2.5.6 The 'const_cast'-operator

The const_cast<type>(expression) operator is used to do away with the const-ness of a (pointer) type. Assume that a function string_op(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));
```

### 2.5.7 The 'reinterpret_cast'-operator

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

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

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

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

### 2.5.8 The void argument list

A function prototype with an empty argument list, such as

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

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

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

Because **C++** maintains strict type checking, an empty argument list is interpreted as the absence of any parameter. The keyword `void` can then be left out. In **C++** the above two declarations are equivalent.

### 2.5.9 The #define __cplusplus

Each **C++** compiler which conforms to the ANSI 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.

### 2.5.10 The usage of standard C functions

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

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

```
extern "C" void *xmalloc(unsigned size);
```

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

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

```
extern "C"
{
    .
    . (declarations)
    .
}
```

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

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

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

### 2.5.11 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
.
. (the declaration of C-functions occurs
.  here, e.g.:)
extern void *xmalloc(unsigned size);
.
#ifdef __cplusplus
}
#endif
```

Using this setup, a normal **C** header file is enclosed by `extern "C" {` which occurs at the start of the file and by }, which occurs at the end of the file. The `#ifdef` directives test for the type of the compilation: **C** or **C++**. The 'standard' header files, such as `stdio.h`, are built in this manner and therefore usable for both **C** and **C++**.

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

```
#ifndef _MYHEADER_H_
#define _MYHEADER_H_
.
. (the declarations of the header file follow here,
.  with #ifdef _cplusplus etc. directives)
.
#endif
```

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

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

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

Apart from all this, the custom has evolved to give **C** header files the extension `.h`, and to give `C++` header files *no* extension. For example, the standard *iostreams* `cin`, `cout` and `cerr` are available after inclusing 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 (yet).

There is more to be said about header files. In section 5.7 the preferred organization of header files when **C++** classes are used is discussed.

### 2.5.12 The definition of local variables

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

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

```
#include <stdio.h>

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

In this code fragment the variable `i` is created inside the `for` statement. According to the ANSI-standard, the variable does not exist prior to the `for`-statement and not beyond the `for`-statement. With some compilers, the variable continues to exist after the execution of the `for`-statement, but a warning like

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

will be issued when the variable is used outside of the `for`-loop. The implication seems clear: define a variable just before the `for`-statement if it's to be used beyond that statement, otherwise the variable can be defined at the `for`-statement itself.

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

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

### 2.5.13 Function Overloading

In **C++** it is possible to define several functions with the same name, performing different actions. The functions must only differ in their argument lists. An example is given below:

```
#include <stdio.h>

void show(int val)
```

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

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

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

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

    return (0);
}
```

In the above fragment three functions `show()` are defined, which only differ in their argument lists: `int`, `double` and `char *`. The functions have the same name. The definition of several functions with the same name is called 'function overloading'.

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

A few remarks concerning function overloading are:

- The usage of more than one function with the same name but quite different actions should be avoided. In the example above, the functions `show()` are still somewhat related (they print information to the screen).

  However, it is also quite possible to define two functions `lookup()`, one of which would find a name in a list while the other would determine the video mode. In this case the two functions have nothing in common except for their name. It would therefore be more practical to use names which suggest the action; say, `findname()` and `getvidmode()`.

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

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

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

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

```
show(0);
```

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

### 2.5.14 Default function arguments

In **C++** it is possible to provide 'default arguments' when defining a function. These arguments are supplied by the compiler when not specified by the programmer.

An example is shown below:

```
#include <stdio.h>

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

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

    showstring();             // in fact this says:
                              // showstring("Hello World!\n");
    return (0);
}
```

The possibility to omit arguments in situations where default arguments are defined is just a nice touch: the compiler will supply the missing argument when not specified. The code of the program becomes by no means shorter or more efficient.

Functions may be defined with more than one default argument:

[4] The return value is, by the way, an integer which states the number of printed characters. This return value is practically never inspected.

```
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

    return (0);
}
```

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

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

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

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

Note that supplying the default arguments in the function definition instead of in the header file would not be the correct approach.

### 2.5.15 The keyword typedef

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

```
struct somestruct
{
    int
        a;
    double
```

```
        d;
    char
        string[80];
};
```

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

```
somestruct
    what;

what.d = 3.1415;
```

### 2.5.16 Functions as part of a struct

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

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

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

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

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

The `point` structure could be used as follows:

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

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

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

The function which 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 `struct`s are used, `->` can be used.

The idea of this syntactical construction is that several types may contain functions with the same name. E.g., a structure representing a circle might contain three `int` values: two values for the coordinates of the center of the circle and one value for the radius. Analogously to the `point` structure, a function `draw()` could be declared which would draw the circle.

# Chapter 3

# A first impression of C++

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

## 3.1 More extensions of C in C++

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

### 3.1.1 The scope resolution operator ::

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

```
#include <stdio.h>

int
    counter = 50;                    // global variable

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

```
    return (0);
}
```

In this code fragment the scope operator is used to address a global variable instead of the local variable with the same name. The usage of the scope operator is more extensive than just this, but the other purposes will be described later.

### 3.1.2 cout, cin and cerr

In analogy to **C**, **C++** defines standard input- and output streams which are opened when a program is executed. The streams are:

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

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

```
#include <iostream>

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

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

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

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

- The streams are declared in the header file `iostream`.
- The streams `cout`, `cin` and `cerr` are in fact 'objects' of a given class (more on classes later), processing the input and output of a program. Note that the term 'object', as used here, means the set of data and functions which defines the item in question.

- The stream `cin` reads data and copies the information to variables (e.g., `ival` in the above example) using the extraction operator >>. We will describe later how operators in **C++** can perform quite different actions than what they are defined to do by the language grammar, such as is the case here. We've seen function overloading. In **C++** *operators* can also have multiple definitions, which is called *operator overloading*.

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

- Special symbolic constants are used for special situations. The termination of a line written by `cout` is realized by inserting the `endl` symbol, rather than using the string `"\n"`.

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

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

- Compared to the standard `C` functions `printf()` and `scanf()`, the usage of the insertion and extraction operators is more `type-safe`. The format strings which are used with `printf()` and `scanf()` can define wrong format specifiers for their arguments, for which the compiler sometimes can't warn. In contrast, argument checking with `cin`, `cout` and `cerr` is performed by the compiler. Consequently it isn't possible to err by providing an `int` argument in places where, according to the format string, a string argument should appear.

- The functions `printf()` and `scanf()`, and other functions which use format strings, in fact implement a mini-language which is interpreted at run-time. In contrast, the `C++` compiler knows exactly `which` in- or output action to perform given which argument.

- The usage of the left-shift and right-shift operators in the context of the streams does illustrate the possibilities of `C++`. Again, it requires a little getting used to, coming from **C**, but after that these overloaded operators feel rather comfortably.

The *iostream library* has a lot more to offer than just `cin, cout` and `cerr`. In chapter 12 *iostreams* will be covered in greater detail.

### 3.1.3 The keyword const

The keyword `const` very often occurs in **C++** programs, even though it is also part of the **C** grammar, where it's much less used.

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

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

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

    return (0);
}
```

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

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

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

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

```
char const *buf;
```

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

In the declaration

```
char *const buf;
```

`buf` itself is a `const` pointer which may not be changed. Whatever `char`s 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 just prior to the keyword may not be changed. 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 handy in cases where confusion may occur. In examples of **C++** code, one often encounters the reverse: `const` *preceding* what should not be altered. That this may result in sloppy code is indicated by our second example above:

```
char const *buf;
```

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

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

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

    return (0);
}
```

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

### 3.1.4 References

Besides the normal declaration of variables, **C++** allows 'references' to be declared as synonyms for variables. A reference to a variable is like an alias; the variable name and the reference name can both be used in statements which affect the variable:

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

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

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

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

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

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

int main()
{
    int
        x;

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

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

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

int main()
{
    int
        x;

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

The way in which **C++** compilers implement references is actually by using pointers: in other words, references in **C++** are just ordinary pointers, as far as the compiler is concerned. However, the programmer does not need to know or to bother about levels of indirection.[1]

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

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

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

```
void some_func(int val)
{
    printf("%d\n", val);
}

int main()
{
    int
        x;

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

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

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

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

int main ()
{
    int
        x;

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

[1]Compare this to the Pascal way: an argument which is declared as `var` is in fact also a pointer, but the programmer needn't know.

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

  Consider the following example:

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

Person
   person[50];                      // database of persons

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

Person const &getperson(int index)  // get a person by indexvalue
{
    ...
    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 function
    printperson(getperson(5));      // references, not copies
                                    // are passed here
    return (0);
}
```

- It should furthermore be noted here that there is another reason for using references when passing objects as function arguments: when passing a reference to an object, the activation of a copy constructor is avoided. We have to postpone this argument to chapter 6

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

```
int &func()
{
    static int
        value;

    return (value);
}
```

This allows the following constructions:

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

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

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

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

  ```
  int &ref;
  ```

  is not allowed; what would `ref` refer to?
- References can, however, be declared as `external`. These references were initialized elsewhere.
- Reference 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.
- Reference may be used as data members of classes. We will return to this usage later.
- In contrast, pointers are variables by themselves. They point at something concrete or just “at nothing”.
- References are aliases for other variables and cannot be re-aliased to another variable. Once a reference is defined, it refers to its particular variable.

- In contrast, pointers can be reassigned to point to different variables.
- When an address-of operator `&` is used with a reference, the expression yields the address of the variable to which the reference applies. In contrast, ordinary pointers are variables themselves, so the address of a pointer variable has nothing to do with the address of the variable pointed to.

## 3.2 Functions as part of structs

The first chapter described that functions can be part of `struct`s (see section 2.5.16). Such functions are called *member functions* or *methods*. This section discusses the actual definition of such functions.

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

```
struct person
{
    char
        name [80],
        address [80];
    void
        print (void);
};
```

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

```
void person::print()
{
    printf("Name:      %s\n"
           "Address:   %s\n", name, address);
}
```

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.

The usage of this `struct` could be, e.g.:

```
person
    p;
```

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

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

## 3.3 Several new data types

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

### 3.3.1 The 'bool' data type

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

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

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

    // from bool to int:

intValue = boolValue ? 1 : 0;
```

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

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

The `bool` data type is found in other programming languages as well. **Pascal** has its type `Boolean`, and **Java** has a `boolean` type. Different from these languages, **C++**'s type `bool` acts like a kind of `int` type: it's primarily a documentation-improving type, having just two values `true` and `false`.

Actually, these values can be interpreted as `enum` values for `1` and `0`. Doing so would neglect the philosophy behind the `bool` data type, but nevertheless: assigning `true` to an `int` variable neither produces warnings nor errors.

Using the `bool`-type is generally more intuitively clear than using `int`. Consider the following prototypes:

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

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

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

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

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

### 3.3.2 The 'wchar_t' data type

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

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

## 3.4 Data hiding: public, private and class

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

**C++** has two special keywords which are concerned with data hiding: `private` and `public`. These keywords can be inserted in the definition of a `struct`. The keyword `public` defines all subsequent fields of a structure as accessible by all code; the keyword `private` defines all subsequent fields as only accessible by the code which is part of the `struct` (i.e., only accessible for the member functions)[2]. In a `struct` all fields are `public`, unless explicitly stated otherwise.

With this knowledge we can expand the `struct person`:

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

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

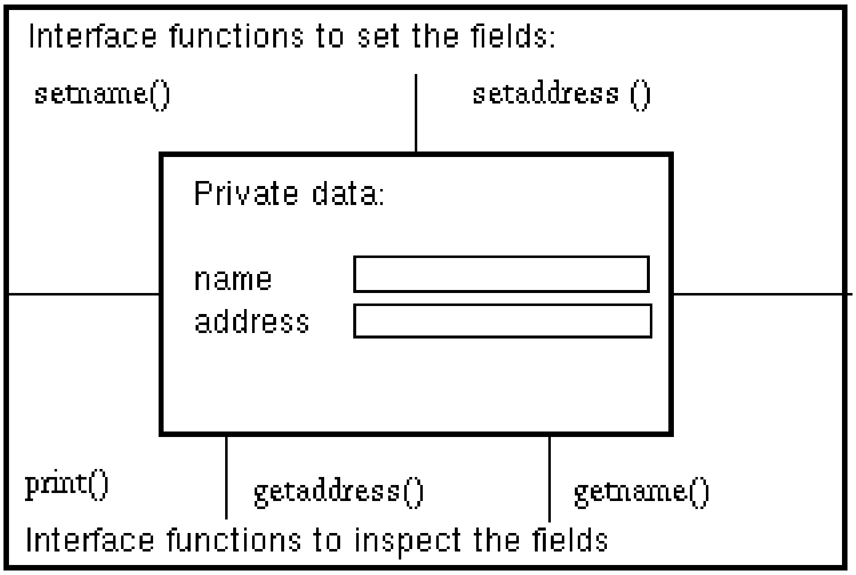
```
person
    x;

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

The concept of data hiding is realized here in the following manner. The actual data of a `struct person` are named only in the structure definition. The data are accessed by the outside world by special functions, which are also part of the definition. These member functions control all traffic between the data fields and other parts of the program and are therefore also called 'interface' functions. The data hiding which is thus realized is illustrated further in figure 3.1.

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

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

[2]Besides `public` and `private`, C++ defines the keyword `protected`. This keyword is not often used and it is left for the reader to explore.




\

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

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

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

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

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

Though data hiding can be realized with `struct`s, more often (almost always) classes are used instead. A `class` is in principle equivalent to a `struct` except that unless specified otherwise, all members (data or functions) are `private`. As far as `private` and `public` are concerned, a `class` is therefore the opposite of a `struct`. The definition of a `class person` would therefore look exactly as shown above, except for the fact that instead of the keyword `struct`, `class` would be used. Our typographic suggestion for class names is a capital as first character, followed by the remainder of the name in lower case (e.g., `Person`).

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

At the end of this chapter we would like to illustrate the analogy between **C** and **C++** as far as `struct`s are concerned. In **C** it is common to define several functions to process a `struct`, which then require a pointer to the `struct` as one of their arguments. A fragment of an imaginary **C** header file is given below:

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

// some functions to manipulate PERSON_ structs
```

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

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

// etc..
```

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

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

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

```
PERSON_
    x;

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

becomes in **C++**:

```
Person
    x;

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

## 3.6 Namespaces

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

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

### 3.6.1 Defining namespaces

Namespaces are defined according to the following syntax:

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

The identifier used in the definition of a namespace is a standard **C++** identifier.

Within the *declarative region*, introduced in the above code example, functions, variables, structs, classes and even (nested) namespaces can be defined or declared. Namespaces cannot be defined within a block. So it is not possible to define a namespace within, e.g., a function. However, it is possible to define a namespace using multiple *namespace* declarations. Namespaces are said to be *open*. This means that a namespace `CppAnnotations` could be defined in a file `file1.cc` and also in a file `file2.cc`. The entities defined in the `CppAnnotations` namespace of files `file1.cc` and `file2.cc` are then united in one `CppAnnotations` namespace region. For example:

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

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

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

Namespace entities can also be defined outside of their namespaces. This topic is discussed in section 3.6.4.

#### Declaring entities in namespaces

Instead of *defiing* entities in a namespace, entities may also be *declared* in a namespace. This allows us to put all the declarations of a namespace in a header file which can thereupon be included in sources in which the entities of a namespace are used. Such a header file could contain, e.g.,

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

#### A closed namespace

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

The entities that are defined in the anonymous namespace are accessible the same way as `static` functions and variables in **C**. The `static` keyword can still be used in **C++**, but its use is more dominant in `class` definitions (see chapter 5). In situations where static variables or functions are necessary, the use of the anonymous namespace is preferred.

### 3.6.2 Referring to entities

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

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

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

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

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

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

the function `cos()` will refer to the `cos()` function in the `CppAnnotations` namespace. This implies that the standard `cos()` function, accepting radials, 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)         // this uses CppAnnotations::cos()
        << ::cos(1.5)       // this uses the standard cos() function
        << endl;
    return (0);
}
```

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

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

    int
        value;              // error: value already defined.

    return (0);
}
```

### The using directive

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

```
using namespace CppAnnotations;
```

Following this directive, *all* entities defined in the `CppAnnotations` namespace are uses 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.

### 3.6.3 The standard namespace

Apart from the anonymous namespace, many entities of the runtime available software (e.g., `cout, cin, cerr` and the templates defined in the *Standard Template Library*, see chapter 11) 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 often 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 often encountered. Whether this should be encouraged is subject of some dispute. Long `using` declarations are of course inconvenient too. So as a rule of thumb one might decide to stick to `using` declarations, up to the point where the list becomes impractically long, at which point a `using` directive could be considered.

### 3.6.4 Nesting namespaces and namespace aliasing

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

At every `using` directive all entities of that namespace can be used without any further prefix. If a namespace is nested, then that namespace can also be used without any further prefix. However, the entities defined in the nested namespace still need the nested namespace's name. Only by using a `using` declaration or directive the qualified name of the nested namespace can be omitted.

When fully qualified names are somehow preferred, while the long form (like `CppAnnotations::Virtual::pointer`) is at the same time considered too long, a namespace *alias* can be used:

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

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

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

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

### Defining entities outside of their namespaces

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

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

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

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

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

        typedef int INT8[8];

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

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

            return (ip);
        }
    }
}
```

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

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

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

        typedef int INT8[8];

        INT8 *funny();
    }
}

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

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

    return (ip);
}
```

At the final code fragment note the following:

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

Finally, note that the function could also have been defined in the `CppAnnotations` region. It that case the `Virtual` namespace would have been required for the function name and its returntype, while the internals of the function would remain the same:

```
namespace CppAnnotations
{
    namespace Virtual
    {
```

```
        void
            *pointer;

        typedef int INT8[8];

        INT8 *funny();
    }

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

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

        return (ip);
    }
}
```

# Chapter 4

# The 'string' data type

**C++** offers a large number of facilities to implement solutions for common problems. Most of these facilities are part of the *Standard Template Library* or they are implemented as *generic algorithms* (see chapter 11).

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

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

Actually, `string` objects are *class type* variables, and the `class` is introduced for the first time in chapter 5. However, in order to use a string, it is not necessary to know what a class is. In this section the operators that are available for strings and some other operations are discussed. The operations that can be performed on strings take the form

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

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

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

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

## 4.1 Operations on strings

Some of the operations that can be performed on strings return indices within the strings. Whenever such an operation fails to find an appropriate index, the *value* `string::npos` is returned.

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

Note that in all operations where `string` objects can be used as arguments, `char const *` values and variables can be used as well.

Some `string`-member functions use *iterators*. Iterators will be covered in section 11.1. The member functions that use iterators are listed in the next section (4.2), they are not further illustrated below.

The following operations can be performed on strings:

- 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 does not use the function argumentlist notation.

```
#include <string>

int main()
{
    string
        stringOne("Hello World"),   // using plain ascii-Z
        stringTwo(stringOne),       // using another string object
        stringThree;                // implicit initialization to ""
                                    // do not use: stringThree();
    return (0);
}
```

- 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 characterstring as its right-hand argument:

```
#include <string>

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

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

    return (0);
}
```

- 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>
```

```
int main()
{
    string
        stringOne("Hello World");
    char const
        *Cstring = stringOne.c_str();

    cout << Cstring << endl;

    return (0);
}
```

- The individual elements of a string object can be reached for reading or writing. For this operation the subscript-operator (`[]`) is available, but *not* the pointer dereferencing operator (∗). The subscript operator does not perform range-checking. If range-checking is required, the `at()` member function can be used instead of the subscript-operator:

```
#include <string>

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

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

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

    // Now using the at() memberfunction:

    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.

- Two strings can be compared for (in)equality or ordering, using the `==`, `!=`, <, <=, > and >= operators or the `compare()` member function can be used. The `compare()` member function comes in different flavors, the plain one (having another `string` object as argument) offers a bit more information than the operators do. The returnvalue of the `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 alphabet (based on the standard ascii-characterset) than the string stored in the string object passed as argument to the `compare()` member function.

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

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

    if (stringOne != stringTwo)
        stringTwo = stringOne;

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

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

    // 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";

    return (0);
}
```

There is no member function to perform a case insensitive comparison of strings.

Overloaded forms of the `compare()` member function have one or two extra arguments.

- If the `compare()` member function is used with two arguments, then the second argument is an index position in the current `string`-object. It indicates the index position in the current `string` object where the comparison should start.
- If the `compare()` member function is used with three arguments, then the third argument indicates the number of characters that should be compared.

See the following example for further details about the `compare()` function.

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

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

```
    // comparing from a certain offset in stringOne

    if (!stringOne.compare("ello World", 1))
        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"

    if (!stringOne.compare("World and more", 6, 5))
        cout << "comparing 'Hello World' from index 6 over 5 positions"
                " to 'World and more': ok\n";

    // The same, but this fails, as all of the chars in stringOne
    // starting at index 6 are compared, not just 3 chars.
    // number of characters in "World and more"

    if (!stringOne.compare("World and more", 6, 3))
        cout << "comparing 'Hello World' from index 6 over 3 positions"
                " to 'World and more': ok\n";
    else
        cout << "Unequal (sub)strings\n";


    return (0);
}
```

- A `string` can be appended to another string. For this the `+=` operator can be used, as well as the `append()` member function. Like the `compare()` function, the `append()` member function may have two 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>

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

    stringOne += " " + stringTwo;

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

    // append only 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;

    return (0);
}
```

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";
```

- So, the `append()` member function is used to append characters at the end of a `string`. It is also possible to insert characters somewhere within a `string`. For this the member function `insert()` is available.

  The `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 <iostream>
#include <string>

int main()
{
    string
        stringOne("Hell ok.");

    stringOne.insert(4, "o ");   // Insert "o " at position 4

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

                                // insert "World" into stringOne
```

```
        stringOne.insert(6, world, 4, 5);

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

        return (0);
    }
```

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

- 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 `replace()` can be used.

  The member function has at least three and possibly five arguments, having the following meanings (see section 4.2 for overloaded versions of `replace()`, using different types of arguments):

  - The first argument indicates the position of the first character that must be replaced
  - The second argument gives the number of characters that must be replaced.
  - The third argument defines the replacement text (a `string` or `char const *`).
  - The fourth argument specifies the index position of the first character in the provided `string`-argument that will be inserted.
  - The fifth argument can be used to specify the number of characters that will be inserted.

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

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

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

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

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

        assert(search != replace);

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

```
                    idx;

                idx = line.find(search);

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

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

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

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

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

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

    stringOne.swap(stringTwo);

    cout << "After: stringOne: " << stringOne << ", stringTwo: "
        << stringTwo << endl;
    return (0);
}
```

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

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

```
#include <string>

int main()
```

```
{
    string
        stringOne("Hello Cruel World");

    stringOne.erase(5, 6);

    cout << stringOne << endl;

    stringOne.erase();

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

    return (0);
}
```

- To find substrings in a `string` the member function `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.
- To extract a substring from a `string` object, the member function `substr()` is available. The returned `string` object contains a copy of the substring in the `string`-object calling `substr()` The member function has two optional arguments:
  - Without arguments, a copy of the `string` itself is returned.
  - The first argument may be used to specify the offset of the first character to be returned.
  - The second argument may be used to specify the number of characters that are to be returned.

  For example:

```
#include <string>

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

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

    return (0);
}
```

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

```
#include <string>
```

```
int main()
{
    string
        line;

    getline(cin, line);

    string::size_type
        pos;

    cout << "Line: " << line << endl
        << "Starting at the first vowel:\n"
        << "'"
            << (
                (pos = line.find_first_of("aeiouAEIOU")) != string::npos ?
                    line.substr(pos)
                :
                    "*** not found ***"
                ) << "'\n"
        << "Starting at the last vowel:\n"
        << "'"
            << (
                (pos = line.find_last_of("aeiouAEIOU")) != string::npos ?
                    line.substr(pos)
                :
                    "*** not found ***"
                ) << "'\n"
        << "Not starting at the first digit:\n"
        << "'"
            << (
                (pos = line.find_first_not_of("1234567890"))
                                                        != string::npos ?
                    line.substr(pos)
                :
                    "*** not found ***"
                ) << "'\n";
    return (0);
}
```

- 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>

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

    cout << "The length of the stringOne string is "
```

```
        << stringOne.size() << " characters\n";

    return (0);
}
```

- If the size of a string is not enough (or if it is too large), the member function `resize()` can be used to make it longer or shorter. Note that operators like + automatically resize the `string` when needed.
- The `size()` member function can be used to determine whether a string holds no characters as well. Alternatively, the `empty()` member function can be used:

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

int main()
{
    string
        stringOne;

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

    stringOne = "";

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


    return (0);
}
```

- The `istream &getline(istream instream, string target, char delimiter)` member function may be used to read a line of text (up to the first delimiter or the end of the stream) from `instream`.

  The delimiter has a default value `'\n'`. It is removed from `instream`, but it is *not* stored in `target`. The function `getline()` was used in several earlier examples (e.g., with the replace() member function).

## 4.2 Overview of operations on strings

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

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

With member functions the types of the parameters are given in a function-prototypical way. With several member functions *iterators* are used. At this point in the Annotations it's a bit premature to discuss iterators, but for referential purposes they have to be mentioned nevertheless. So, a forward reference is used here: see section 11.1 for a more detailed discussion of *iterators*.

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

### 4.2.1 The string-initializers

- `string object`:

  Initializes `object` to an empty string.

- `string object(string::size_type n, char c)`:

  Initializes `object` with `n` characters `c`.

- `string object(string argument)`:

  Initializes `object` with `argument`.

- `string object(string argument, string::size_type idx, string::size_type n = pos)`:

  Initializes `object` with `argument`, using `n` characters of `argument`, starting at index `idx`.

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

  Initializes `object` with the range of characters implied by the provided `InputIterators`.

### 4.2.2 The `string`-iterators

See section 11.1 for details about *iterators*.

- Forward iterators:
  - `begin()`
  - `end()`
- Reverse iterators:
  - `rbegin()`
  - `rend()`

### 4.2.3 The `string`-operators

- `object = argument`.

  Assignment of `argument` to `object`. May also be used for initializing `string` objects.

- `object = c`.

Assignment of `char c` to `object`. May *not* be used for initializing `string` objects.

- `object += argument`.

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

- `argument1 + argument2`.

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

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

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

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

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

- `argument1 == argument2`.

  The equality operator may be used to compare a `string` object to another `string` or `char const *` value. The operator `!=` is available as well. The returnvalue is a `bool`, which is `true` if the two strings are equal (i.e., contain the same characters). `!=` returns `false` in that case.

- `argument1 < argument2`.

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

- `ostream stream; stream << argument`.

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

- `istream stream; stream >> object`.

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

### 4.2.4 The string member functions

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

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

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

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

- `string &object.append(string argument, string::size_type pos = 0; string::size_type n = string::npos)`:

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

  If `argument` is of type `char const *`, parameter `pos` 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 `object`.

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

- `string &object.assign(string argument, string::size_type pos = 0; string::size_type n = string::npos)`:

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

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

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

- `string::size_type object.capacity()`:

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

- `int object.compare(string argument, string::size_type pos, string::size_type n)`:

  This member function may be used to compare (according to the ascii-character set) the strings stored in `object` and `argument`. The parameter `n` may be used to specify the number of characters in `argument` that are used in the comparison, the parameter `pos` may be used to specify the initial character in `object` that is used in the comparison.

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

If the third argument is omitted, the first n characters of object are copied to argument. If the thirst argument is given, copying starts from &object[pos]. Following the copying, no ascii-Z is appended to the copied string. If n exceeds object.length(), at most object.length() characters are copied. The actual number of characters that were copied is returned.

- `char const *object.c_str`:

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

- `char const *object.data()`:

  returns the raw text stored in object.

- `bool object.empty()`:

  returns true if object contains no data.

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

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

  - If pos is specified, the contents of object are erased from index position pos until the end of object.
  - If pos and n are provided, n characters of object, starting at index position pos are erased.

- `iterator object.erase(iterator p)`:

  The contents of object are erased until (iterator) position p. The iterator p is returned.

- `iterator object.erase(iterator f, iterator l)`:

  The range of characters of object, implied by the iterators f and l are erased. The iterator f is returned.

- `string::string::size_type object.find(string argument, string::size_type pos)`:

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

- `string::size_type object.find(char const *argument, string::size_type pos, string::size_type n)`:

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

- `string::size_type object.find(char c, string::size_type pos)`:

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

- `string::size_type object.find_first_of(string argument, string::size_type pos)`:

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

- `string::size_type object.find_first_of(char const* argument, string::size_type pos, string::size_type n)`:

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

- `string::size_type object.find_first_of(char c, string::size_type pos)`:

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

- `string::size_type object.find_first_not_of(string argument, string::size_type pos)`:

  Returns the index in `object` where a character not appearing in `argument` is found. If the argument `pos` is omitted, the search starts at the beginning of `object`. If provided, it refers to the index in `object` where the search for `argument` should start.

- `string::size_type object.find_first_not_of(char const* argument, string::size_type pos, string::size_type n)`:

  Returns the index in `object` where any character not appearing in `argument` is found. The parameter `n` indicates the number of characters of `object` that should be used in the search: it defines a partial string starting at the beginning of `object`. If omitted, all characters in `object` are used. The parameter `pos` refers to the index in `object` where the search for `argument` should start. If the parameter `pos` is omitted as well, `object` is scanned completely.

- `string::size_type object.find_first_not_of(char c, string::size_type pos)`:

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

- `string::size_type object.find_last_of(string argument, string::size_type pos)`:

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

- `string::size_type object.find_last_of(char const* argument, string::size_type pos, string::size_type n)`:

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

- `string::size_type object.find_last_of(char c, string::size_type pos)`:

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

- `string::size_type object.find_last_not_of(string argument, string::size_type pos)`:

  Returns the last index in `object` where any character not appearing in `argument` is found. If the argument `pos` is omitted, the search starts at the beginning of `object`. If provided, it refers to the index in `object` where the search for `argument` should start.

- `string::size_type object.find_last_not_of(char const* argument, string::size_type pos, string::size_type n)`:

  Returns the last index in `object` where any character not appearing in `argument` is found. The parameter `n` indicates the number of characters of `object` that should be used in the search: it defines a partial string starting at the beginning of `object`. If omitted, all characters in `object` are used. The parameter `pos` refers to the index in `object` where the search for `argument` should start. If the parameter `pos` is omitted as well, all of `object` is scanned.

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

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

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

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

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

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

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

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

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

- `iterator object.insert(iterator p, char c)`:

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

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

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

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

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

- `string::size_type object.length()`:

  returns the number of characters stored in `object`.

- `string::size_type object.max_size()`:

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

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

  The substring of `n1` characters of `object`, starting at position `pos1` is replaced by `argument`. If `n1` is set to 0, the member function *inserts* `argument` into `object`. The basic form uses `argument` completely. The way other forms of `replace()` work depends on the specification of extra arguments:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- `string::size_type object.size()`:

  returns the number of characters stored in `object`.

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

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

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

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

# Chapter 5

# Classes

In this chapter classes are the topic of discussion. Two special member functions, the constructor and the destructor, are introduced.

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

Let's start off by introducing the declaration of a `class Person` right away. The class declaration is normally contained in the *header file* of the class, e.g., `person.h`. The class declaration is generally not called a *declaration*, though. Rather, the common name for class declarations is *class interface*, to be distinguished from the definitions of the function members, called the *class implementation*. Thus, the interface of the `class Person` is given next:

```
class Person
{
    public:                 // interface functions
        void setname(char const *n);
        void setaddress(char const *a);
        void setphone(char const *p);

        char const *getname(void);
        char const *getaddress(void);
        char const *getphone(void);

    private:                // data fields
        char *name;         // name of person
        char *address;      // address field
        char *phone;        // telephone number
};
```

The data fields in this class are `name`, `address` and `phone`. The fields are `char` $*$s which point to allocated memory. The data are `private`, which means that they can only be accessed by the functions of the class `Person`.

The data are manipulated by interface functions which take care of all communication with code outside of the class. Either to set the data fields to a given value (e.g., `setname()`) or to inspect the data (e.g., `getname()`).

Note once again how similar the `class` is to the `struct`. The fundamental difference being that by default classes have *private* members, whereas structs have *public* members. Since the convention calls for the public members of a class to appear first, the keyword `private` is needed to switch back from public members to the (default) private situation.

## 5.1 Constructors and destructors

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 basic forms and functions of these two categories are discussed next.

### 5.1.1 The constructor

The constructor member function has by definition the same name as the corresponding class. The constructor has *no return value* specification, not even `void`. E.g., for the class `Person` the constructor is `Person::Person()`. The **C++** run-time system makes sure that the constructor of a class, if defined, is called when an object of the class is created. It is of course possible to define a class which has no constructor at all; in that case the run-time system either calls no function or it calls a dummy constructor (i.e., a constructor which performs no actions) when a corresponding object is created. The actual generated code of course depends on the compiler[1].

Objects may be defined at a local (function) level, or at a global level (in which its status is comparable to a global variable.

When an object is a local (non-static) variable of a function, the constructor is called every time the function is called at the point where the variable is defined (a subtlety here is that a variable may be defined implicitly as, e.g., a temporary variable in an expression).

When an object is a static variable, the constructor is called when the function in which the static variable is defined is called for the first time.

When an object is a global variable the constructor is called when the program starts. Note that in even this case the constructor is called even before the function `main()` is started. This feature is illustrated in the following listing:

```
#include <iostream>

// a class Test with a constructor function
class Test
{
    public:                 // 'public' function:
        Test();             // the constructor
```

[1]A compiler-supplied constructor in a class which contains composed objects (see section 5.5) will 'automatically' call the member initializers, and therefore does perform some actions. We postpone the discussion of such constructors to 5.5.1.

```
};

Test::Test()                // here is the
{                           // definition
    cout << "constructor of class Test called\n";
}

// and here is the test program:
Test
    g;                      // global object

void func()
{
    Test                    // local object
        l;                  // in function func()

    cout << "here's function func()" << endl;
}

int main()
{
    Test                    // local object
        x;                  // in function main()

    cout << "main() function" << endl;
    func();
    return (0);
}
```

The listing shows how a class `Test` is defined which consists of only one function: the constructor. The constructor performs only one action; a message is printed. The program contains three objects of the class `Test`: one global object, one local object in `main()` and one local object in `func()`.

Concerning the definition of a constructor we have the following remarks:

- The constructor has the same name as its class.
- The constructor may not be defined with a return value. This is true for the declaration of the constructor in the class definition, as in:

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

  and also holds true for the definition of the constructor function, as in:

```
/* no return value here */ Test::Test()
{
    // statements ...
}
```

- The constructor function in the example above has no arguments. Therefore it is also called the *default constructor*. That the function has no arguments is, however, no requirement *per se*. We shall later see that it is possible to define constructors with arguments. Once a constructor function is defined explicitly, the default constructor doesn't exist anymore, unless the default constructor is definied explicitly itself.

The constructor of the three objects of the class `Test` in the above listing are called in the following order:

- The constructor is first called for the global object `g`.
- Next the function `main()` is started. The object `x` is created as a local variable of this function and hence the constructor is called again. After this we expect to see the text `main() function`.
- Finally the function `func()` is activated from `main()`. In this function the local object `l` is created and hence the constructor is called. After this, the message `here's function func()` appears.

As expected, the program yields therefore the following output (the text in parentheses is added for illustration purposes):

```
constructor of class Test called        (global object g)
constructor of class Test called        (object x in main())
main() function
constructor of class Test called        (object l in func())
here's function func()
```

### 5.1.2 The destructor

The second special member function is the destructor. This function is the opposite of the constructor in the sense that it is invoked when an object ceases to exist. For objects which are local non-static variables, the destructor is called when the block in which the object is defined is left: the destructors of objects that are defined in nested blocks of functions are therefore usually called before the function itself terminates. The destructors of objects that are defined somewhere in the outer block of a function are called just before the function returns (terminates). For static or global variables the destructor is called before the program terminates.

However, when a program is interrupted using an `exit()` call, the destructors are called *only* for global objects which exist at that time. Destructors of objects defined *locally* within functions are not called when a program is forcefully terminated using `exit()`.

When defining a destructor for a given class the following rules apply:

- The destructor function has the same name as the class but prefixed by a tilde.
- The destructor has neither arguments nor a return value.

The destructor for the class `Test` from the previous section could be declared as follows:

```
class Test
{
    public:
        Test();                     // constructor
        ~Test();                    // destructor
        // any other members
};
```

The position of the constructor(s) and destructor in the class definition is dictated by convention: First the constructors are declared, then the destructor, and only then any other members follow.

### 5.1.3 A first application

One of the applications of constructors and destructors is the management of memory allocation. This is illustrated using the class `Person`.

As illustrated at the beginning of this chapter, the class `Person` contains three `private` pointers, all `char` *s. These data members are manipulated by the interface functions. The internal workings of the class are as follows: when a name, address or phone number of a `Person` is defined, memory is allocated to store these data. An obvious setup is described below:

- The constructor of the class makes sure that the data members are initially 0-pointers.
- The destructor releases all allocated memory.
- The defining of a name, address or phone number (by means of the `set...()` functions) consists of two steps. First, previously allocated memory is released. Next, the string which is supplied as an argument to the `set...()` function is duplicated in memory.
- Inspecting a data member by means of one of the `get...()` functions simply returns the corresponding pointer: either a 0-pointer, indicating that the data is not defined, or a pointer to allocated memory holding the data.

The `set...()` functions are illustrated below. Strings are duplicated in this example by an imaginary function `xstrdup()`, which would duplicate a string or terminate the program when the memory pool is exhausted[2].

[2]As a word to the initiated reader it is noted here that many other ways to handle the memory allocation are possible here: As discussed in section 6, `new` could be used, together with `set_new_handler()`, or exceptions could be used to catch any failing memory allocation. However, since we haven't covered that subject yet, and since these annotations start from **C**, we used the tried and true method of a 'protected allocation function' `xstrdup()` here for didactical reasons.

```
// interface functions set...()
void Person::setname(char const *n)
{
    free(name);
    name = xstrdup(n);
}

void Person::setaddress(char const *a)
{
    free(address);
    address = xstrdup(a);
}

void Person::setphone(char const *p)
{
    free(phone);
    phone = xstrdup(p);
}
```

Note that the statements `free(...)` in the above listing are executed unconditionally. This never leads to incorrect actions: when a name, address or phone number is defined, the corresponding pointers point to previously allocated memory which should be freed. When the data are not (yet) defined, then the corresponding pointer is a 0-pointer; and `free(0)` performs no action[3].

Furthermore it should be noted that this code example uses the standard `C` function `free()` which should be familiar to most `C` programmers. The **`delete`** statement, which has more '`C++` flavor', will be discussed later.

The interface functions `get...()` are defined now. Note the occurence of the keyword `const` following the parameter lists of the functions: the member functions are *const member functions*, indicating that they will not modify their object when they're called. The matter of *const member functions* is postponed to section 5.2, where it will be discussed in greater detail.

```
// interface functions get...()
char const *Person::getname() const
{
    return (name);
}

char const *Person::getaddress() const
{
   return (address);
}

char const *Person::getphone() const
{
   return (phone);
```

[3]Actually, free(0) should perform no action. However, later on we'll introduce the operators `new` and `delete`. With the `delete` operator `delete 0` is formally ignored.

```
}
```

The destructor, constructor and the class definition are given below.

```
// class definition
class Person
{
    public:
        Person();          // constructor
        ~Person();         // destructor

        // functions to set fields
        void setname(char const *n);
        void setaddress(char const *a);
        void setphone(char const *p);

        // functions to inspect fields
        char const *getname() const;
        char const *getaddress() const;
        char const *getphone() const;

    private:
        char *name;             // name of person
        char *address;          // address field
        char *phone;            // telephone number
};

// constructor
Person::Person()
{
    name = 0;
    address = 0;
    phone = 0;
}

// destructor
Person::~Person()
{
    free(name);
    free(address);
    free(phone);
}
```

To demonstrate the usage of the class `Person`, a code example follows next. An object is initialized and passed to a function `printperson()`, which prints the contained data. Note also the usage of the reference operator `&` in the argument list of the function `printperson()`. This way only a reference to a `Person` object is passed, rather than a whole object. The fact that `printperson()` does not modify its argument is evident from the fact that the argument is declared `const`. Also note that the example doesn't show where the destructor is called; this action occurs implicitly when the below function `main()` terminates and hence when its local variable `p` ceases to exist.

It should also be noted that the function `printperson()` could be defined as a `public` member function of the class `Person`.

```
#include <iostream>

void printperson(Person const &p)
{
    cout << "Name    : " << p.getname() << endl
         << "Address : " << p.getaddress() << endl
         << "Phone   : " << p.getphone() << endl;
}

int main()
{
    Person
        p;

    p.setname("Linus Torvalds");
    p.setaddress("E-mail: Torvalds@cs.helsinki.fi");
    p.setphone(" - not sure - ");

    printperson(p);
    return (0);
}
```

When `printperson()` receives a fully defined `Person` object (i.e., containing a name, address and phone number), the data are correctly printed. However, when a `Person` object is only partially filled, e.g. with only a name, `printperson()` passes 0-pointers to `cout`. This unesthetic feature can be remedied with a little more code:

```
void printperson(Person const &p)
{
    if (p.getname())
        cout << "Name   : " << p.getname() << "\n";
    if (p.getaddress())
        cout << "Address : " << p.getaddress() << "\n";
    if (p.getphone())
        cout << "Phone  : " << p.getphone() << "\n";
}
```

Alternatively, the constructor `Person::Person()` might initialize the members to 'printable defaults', like `" ** undefined ** "`.

### 5.1.4 Constructors with arguments

In the above declaration of the class `Person` the constructor has no arguments. **C++** allows constructors to be defined with argument lists. The arguments are supplied when an object is created.

For the class `Person` a constructor may be handy which expects three strings: the name, address and phone number. Such a constructor is shown below:

```
Person::Person(char const *n, char const *a, char const *p)
{
    name = xstrdup(n);
    address = xstrdup(a);
    phone = xstrdup(p);
}
```

The constructor must be included in the class declaration, as illustrated here:

```
class Person
{
    public:
        Person::Person(char const *n,
            char const *a, char const *p);
        .
        .
        .
};
```

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.

The usage of a constructor with arguments is illustrated in the following code fragment. The object `a` is initialized at its definition:

```
int main()
{
    Person
        a("Karel", "Rietveldlaan 37", "542 6044"),
        b;

    return (0);
}
```

In this example, the `Person` objects `a` and `b` are created when `main()` is started. For the object `a` the constructor with arguments is selected by the compiler. For the object `b` the default constructor (without arguments) is used.

### The order of construction

The possibility to pass arguments to constructors offers us the chance to monitor at which exact moment in a program's execution an object is created or destroyed. This is shown in the next listing, using a class `Test`:

```
class Test
{
    public:
        // constructors:
        Test();                     // argument-free
        Test(char const *name);     // with a name argument
        // destructor:
        ~Test();

    private:
        // data:
        char *n;                     // name field
};

Test::Test()
{
    n = xstrdup("without name");
    printf("Test object without name created\n");
}

Test::Test(char const *name)
{
    n = xstrdup(name);
    cout << "Test object " << name << " created" << endl;
}

Test::~Test()
{
    cout << "Test object " << n << " destroyed" << endl;
    free(n);
}
```

By defining objects of the class `Test` with specific names, the construction and destruction of these objects can be monitored:

```
Test
    globaltest("global");

void func()
{
    Test
        functest("func");
```

```
}

int main()
{
    Test
        maintest("main");

    func();
    return (0);
}
```

This test program thus leads to the following (and expected) output:

```
Test object global created
Test object main created
Test object func created
Test object func destroyed
Test object main destroyed
Test object global destroyed
```

## 5.2 Const member functions and const objects

The keyword `const` is often seen in the declarations of member functions following the argument list. This keyword is used to indicate that a member function does not alter the data fields of its object, but only inspects them. Using the example of the class `Person`, the `get...()` functions should be declared `const`:

```
class Person
{
    public:
        .
        .
        // functions to inspect fields
        char const *getname(void) const;
        char const *getaddress(void) const;
        char const *getphone(void) const;

    private:
        .
        .
};
```

As is illustrated in this fragment, the keyword `const` occurs *following* the argument list of functions. Note that in this situation the rule of thumb given in section 3.1.3 applies once again: whichever appears **before** the keyword `const`, may not be altered and doesn't alter (its own) data.

The same specification must be repeated in the definition of member functions themselves:

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

A member function which is declared and defined as `const` may not alter any data fields of its class. In other words, a statement like

```
name = 0;
```

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

The `const` member functions exist because **C++** allows `const` objects to be created, or references to `const` objects to be passed on 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 variable is created, but subsequent assignments cannot take place.

The following example shows how a `const` object of the class `Person` can be defined. When the object is created the data fields are initialized by the constructor:

```
Person const
    me("Karel", "karel@icce.rug.nl", "542 6044");
```

Following this definition it would be illegal to try to redefine the name, address or phone number for the object `me`: a statement as

```
me.setname("Lerak");
```

would not be accepted by the compiler. Once more, look at the position of the `const` keyword in the variable definition: `const`, following `Person` and preceding `me` associates to the left: the `Person` object in general must remain unaltered. Hence, if multiple objects were defined here, both would be constant `Person` objects, as in:

```
Person const        // all constant Person objects
```

```
        kk("Karel", "karel@icce.rug.nl", "542 6044"),
        fbb("Frank", "frank@icce.rug.nl", "403 2223");
```

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.

## 5.3 The operators new and delete

The **C++** language defines two operators which are specific for the allocation and deallocation of memory. These operators are `new` and `delete`.

The most basic example of the use of these operators is given below. An `int` pointer variable is used to point to memory which is allocated by the operator `new`. This memory is later released by the operator `delete`.

```
int
    *ip;

ip = new int;
// any other statements
delete ip;
```

Note that `new` and `delete` are *operators* and therefore do not require parentheses, which are 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).

### 5.3.1 Allocating and deallocating arrays

When the operator `new` is used to allocate an array, the size of the variable is placed between square brackets following the type:

```
int
    *intarr;

intarr = new int [20];    // allocates 20 ints
```

The syntactical rule for the operator `new` is that this operator must be followed by a type, optionally followed by a number in square brackets. The type and number specification lead to an expression which is used by the compiler to deduce its size; in **C** an expression like `sizeof(int[20])` might be used.

An array is deallocated by using the operator `delete`:

```
delete [] intarr;
```

In this statement the array operators [] indicate that an array is being deallocated. The rule of thumb here is: whenever `new` is followed by `[]`, `delete` should be followed by it too.

What happens if `delete` rather than `delete []` is used? Consider the following situation: a `class X` is defined having a destructor telling us that it's called. In a `main()` function an array of two `X` 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>

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

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

int main()
{
    X
      *a;

    a = new X[2];

    cout << "Destruction with []'s" << endl;

    delete [] a;

    a = new X[2];

    cout << "Destruction without []'s" << endl;

    delete a;

    return (0);
}
```

Here's the generated output:

```
Destruction with []'s
```

```
X destructor called
X destructor called
Destruction without [] 's
X destructor called
```

So, as we can see, the destructor of the individual `X` objects are called if the `delete []` syntax is followed, and not if the `[]` is omitted.

If no destructor is defined, it is not called. Consider the following fragment:

```
#include <iostream>

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

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

int main()
{
    X
      **a;

    a = new X* [2];

    a[0] = new X [2];
    a[1] = new X [2];

    delete [] a;

    return (0);
}
```

This program produces no messages 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 as we do not have something as a 'class pointer to X objects'. Consequently, the `[]` is ignored.

Now, because of the `[]` being ignored, not all elements of the array `a` points to are considered when `a` is deleted. The two pointer elements of `a` *are* deleted, though, because `delete a` (note that the `[]` is not written here) frees the memory pointed to by `a`. That's all there is to it.

What if we don't want this, but require the `X` objects pointed to by the elements of `a` to be deleted as well? 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 X objects, which will destroy all elements if the [] operator is used, as in:

```
#include <iostream>

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

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

int main()
{
    X
      **a;

    a = new X* [2];

    a[0] = new X [2];
    a[1] = new X [2];

    for (int index = 0; index < 2; index++)
        delete [] a[index];

    delete a;

    return (0);
}
```

- Define a class containing a pointer to X objects, and allocate a pointer to this super-class, rather than a pointer to a pointer to X objects. The topic of containing classes in classes, *composition*, is discussed in section 5.5.

### 5.3.2 New and delete and object pointers

The operators `new` and `delete` are also used when an object of a given class is allocated. As we have seen in the previous section, the advantage of the operators `new` and `delete` over functions like `malloc()` and `free()` lies in the fact that `new` and `delete` call the corresponding constructors or destructor. This is illustrated in the next example:

```
Person
    *pp;                    // ptr to Person object
```

```
pp = new Person;            // now constructed
...
delete pp;                  // now destroyed
```

The allocation of a new `Person` object pointed to by `pp` is a two-step process. First, the memory for the object itself is allocated. Second, the constructor is called which initializes the object. In the above example the constructor is the argument-free version; it is however also possible to choose an explicit constructor:

```
pp = new Person("Frank", "Oostumerweg 17", "050 403 2223");
...
delete pp;
```

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 deallocate the memory 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 with `new` and `delete`. In this case the size of the array is given between the `[]` when the array is created:

```
Person
    *personarray;

personarray = new Person [10];
```

The compiler will generate code to call the default constructor for each object which is created. As we have seen, the array operator `[]` must be used with the `delete` operator 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. Note again that `delete personarray` would only release the memory of the array itself.

### 5.3.3 The function set_new_handler()

The **C++** run-time system makes sure that when memory allocation fails, an error function is activated. By default this function returns the value 0 to the caller of `new`, so that the pointer which is assigned by `new` is set to zero. The error function can be redefined, but it must comply with a few prerequisites, which are, unfortunately, compiler-dependent. E.g., for the Microsoft `C/C++` compiler version 7, the prerequisites are:

- The function is supplied one argument, a `size_t` value which indicates how many bytes should have been allocated[4].
- The function must return an `int`, which is the value passed by `new` to the assigned pointer.

The Gnu `C/C++` compiler `gcc`, which is present on many Unix platforms, requires that the error handler:

- has no arguments, and
- returns no value (a `void` return type).

Then again, Microsoft's `Visual C++` interprets the returnvalue of the the function as follows:

- The run-time system retries allocation each time the function returns a nonzero value and fails new if the function returns 0.

In short: there's no standard here, so make sure that you lookup the particular characteristics of the `set_new_handler` function for your compiler. Whatever you do, in any case make sure you use this function: it saves you a lot of checks (and problems with a failing allocation that you just happened to forget to protect with a check...).

The redefined error function might, e.g., print a message and terminate the program. The user-written error function becomes part of the allocation system through the function `set_new_handler()`, defined in the header file `new.h`. With some compilers, the installing function is called `_set_new_handler()` (note the leading underscore).

The implementation of an error function is illustrated below. This implementation applies to the `Gnu` C/C++ requirements[5]:

```
#include <new.h>
#include <iostream>

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

int main()
{
    int
        *ip;
    long
        total_allocated = 0;

    // install error function
```

[4]The type `size_t` is usually identical to `unsigned`.

[5]The actual try-out of the program is not encouraged, as it will slow down the computer enormously due to the resulting occupation of *Unix's swap area*

```
    set_new_handler(out_of_memory);

    // eat up all memory
    puts("Ok, allocating..");
    while (1)
    {
        ip = new int [10000];
        total_allocated += 10000 * sizeof(int);
        printf("Now got a total of %ld bytes\n",
                total_allocated);
    }

    return (0);
}
```

The advantage of an allocation error function lies in the fact that once installed, `new` can be used without wondering whether the allocation succeeded or not: upon failure the error function is automatically invoked and the program exits. It is good practice to install a `new` handler in each **C++** program, even when the actual code of the program does not allocate memory. Memory allocation can also fail in not directly visible code, e.g., when streams are used or when strings are duplicated by low-level functions.

Note that it may *not* be assumed that the standard **C** functions which allocate memory, such as `strdup()`, `malloc()`, `realloc()` etc. will trigger the `new` handler when memory allocation fails. This means that once a `new` handler is installed, such functions should not automatically be used in an unprotected way in a **C++** program. As an example of the use of `new` for duplicating a string, a rewrite of the function `strdup()` using the operator `new` is given in section 6. It is strongly suggested to revert to this approach, rather than to using functions like `xstrdup()`, when the allocation of memory is required.

## 5.4 The keyword inline

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

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

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

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

```
puts(frank.getname());
```

the following actions take place:

- The function `Person::getname()` is called.
- This function returns the value of the pointer `name` of the object `frank`.
- This value, which is a pointer to a string, is passed to `puts()`.
- The function `puts()` finally is called and prints a string.

Especially the first part of these actions leads to some time loss, since an extra function call is necessary to retrieve the value of the `name` field. Sometimes a faster process may be desirable, in which the `name` field becomes immediately available; thus avoiding the call to `getname()`. This can be realized by using `inline` functions, which can be defined in two ways.

### 5.4.1 Inline functions within class declarations

Using the first method to implement `inline` functions, the code of a function is defined *in a class declaration itself.* For the class `Person` this would lead to the following implementation of `getname()`:

```
class Person
{
    public:
        ...
        char const *getname(void) const
        {
            return (name);
        }
        ...
};
```

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

Thus, `inline` functions appearing in the class interface show their full (and standard) definition *within* the class interface itself.

The effect of this is the following. When `getname()` is called in a program statement, the compiler *generates the code of the function* when the function is used in the source-text, rather than a call to the function, appearing only once in the compiled program.

This construction, where the function code itself is inserted rather than a call to the function, is called an inline function. Note that the use of inline function results in duplication of the code of the function for each invokation 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.

### 5.4.2 Inline functions outside of class declarations

The second way to implement inline functions leaves a class interface intact, but mentions the keyword `inline` in the function definition. The interface and implementation in this case are as follows:

```
class Person
{
    public:
        ...
        char const *getname(void) const;
        ...
};

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

Again, the compiler will insert the code of the function `getname()` instead of generating a call.

However, the `inline` function must still appear in the same file as the class interface, and cannot be compiled to be stored in, e.g., a library. The reason for this is that the *compiler* rather than the *linker* must be able to insert the code of the function in a source text offered for compilation. Code stored in a library is inaccessible to the compiler. Consequently, inline functions are always defined together with the class interface.

### 5.4.3 When to use inline functions

When should `inline` functions be used, and when not? There is a number of simple rules of thumb which may be followed:

- In general `inline` functions should **not** be used. *Voilà*, that's simple, isn't it?
- Defining `inline` functions can be considered once a fully developed and tested program runs too slowly and shows 'bottlenecks' in certain functions. A profiler, which runs a program and determines where most of the time is spent, is necessary for such optimization.
- `inline` functions can be used when member functions consist of one very simple statement (such as the return statement in the function `Person::getname()`).
- By defining a function as `inline`, its implementation is inserted in the code wherever the function is used. As a consequence, when the *implementation* of the inline function changes, all sources using the inline function must be recompiled. In practice that means that all functions must be recompiled that include (either directly or indirectly) the header file of the class in which the inline function is defined.
- It is only useful to implement an `inline` function when the time which is spent during a function call is long compared to the code in the function. An example where an `inline` function has no effect at all is the following:

```
void Person::printname() const
{
    cout << name << endl;
}
```

This function, which is, for the sake of the argument, presumed to be a member of the class `Person`, contains only one statement.

However, the statement takes a relatively long time to execute. In general, functions which perform input and output take lots of time. The effect of the conversion of this function `printname()` to `inline` would therefore lead to a very insignificant gain in execution time.

All `inline` functions have one disadvantage: the actual code is inserted by the compiler and must therefore be known compile-time. Therefore, as mentioned earlier, an `inline` function can never be located in a run-time library. Practically this means that an `inline` function is placed near the interface of a class, usually in the same header file. The result is a header file which not only shows the **declaration** of a class, but also part of its **implementation**, thus blurring the distinction between interface and implementation.

Finally, note that using the keyword `inline` is not really an *order* for the compiler. Rather, it is a *suggestion* the compiler may either choose to follow or to ignore.

## 5.5 Objects in objects: composition

An often recurring situation is one where objects are used as data fields in class definitions. This is referred to as **composition**.

For example, the class `Person` could hold information about the name, address and phone number, but additionally a class `Date` could be used to keep the information about the birth date:

```
class Person
{
    public:
        // constructor and destructor
        Person();
        Person(char const *nm, char const *adr,
                char const *ph);
        ~Person();

        // interface functions
        void setname(char const *n);
        void setaddress(char const *a);
        void setphone(char const *p);
        void setbirthday(int yr, int mnth, int d);

        char const *getname() const;
        char const *getaddress() const;
        char const *getphone() const;
```

```
        int getbirthyear() const;
        int getbirthmonth() const;
        int getbirthday() const;

    private:
        // data fields
        char *name, *address, *phone;
        Date birthday;
};
```

We shall not further elaborate on the class `Date`: this class could, e.g., consist of three `int` data fields to store a year, month and day. These data fields would be set and inspected using interface functions `setyear()`, `getyear()` etc..

The interface functions of the class `Person` would then use `Date`'s interface functions to manipulate the birth date. As an example the function `getbirthyear()` of the class `Person` is given below:

```
int Person::getbirthyear() const
{
    return (birthday.getyear());
}
```

Composition is not extraordinary or **C++** specific: in **C** it is quite common to include `struct`s or `union`s in other compound types. Note that the composed objects can be reached through their member functions: the normal field selector operators are used for this.

However, the initialization of the composed objects deserves some extra attention: the topics of the coming sections.

### 5.5.1 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 the constructor of the composing class. This is illustrated below for the composed class `Date` in a `Person`. In this fragment it assumed that a constructor for a `Person` should be defined expecting six arguments: the name, address and phone number plus the year, month and day of the birth date. It is furthermore assumed that the composed class `Date` has a constructor with three `int` arguments for the year, month and day:

```
Person::Person(char const *nm, char const *adr,
                char const *ph,
                int d, int m, int y)
:
    birthday(d, m, y)
```

```
{
    name = xstrdup(nm);
    address = xstrdup(adr);
    phone = xstrdup(ph);
}
```

Note that following the argument list of the constructor `Person::Person()`, the constructor of the data field `Date` is specifically called, supplied with three arguments. This constructor is explicitly called for the composed object `birthday`. This occurs even **before** the code block of `Person::Person()` is executed. This means that when a `Person` object is constructed and when six arguments are supplied to the constructor, the `birthday` field of the object is initialized even before `Person`'s own data fields are set to their values.

In this situation, the constructor of the composed data member is also referred to as *member initializer.*

When several composed data members of a class exist, all member initializers can be called using a 'constructor list': this list consists of the constructors of all composed objects, separated by commas.

When member initializers are *not* used, the compiler automatically supplies a call to the *default constructor* (i.e., the constructor without arguments). In this case a default constructor *must* have been defined in the composed class.

Member initializers should be used as much as possible: not using member initializers can result in inefficient code, and can be downright necessary. As an example showing the inefficiency of not using a member initializer, consider the following code fragment where the `birthday` field is not initialized by the `Date` constructor, but instead the `setday()`, `setmonth()` and `setyear()` functions are called:

```
Person::Person(char const *nm, char const *adr,
                char const *ph,
                int d, int m, int y)
{
    name = xstrdup(nm);
    address = xstrdup(adr);
    phone = xstrdup(ph);

    birthday.setday(d);
    birthday.setmonth(m);
    birthday.setyear(y);
}
```

This code is inefficient because:

- first the default constructor of `birthday` is called (this action is implicit),
- and subsequently the desired date is set explicitly by member functions of the class `Date`.

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.

This means that when the definition of a `Person` is changed so that the data member `birthday` is declared as a `const` object, the implementation of the constructor `Person::Person()` with six arguments *must* use member initializers. Calling the `birthday.set...()` would be illegal, since these are no `const` functions.

Concluding, the rule of thumb is the following: when composition of objects is used, the member initializer method is preferred to explicit initialization of the composed object. This not only results in more efficient code, but it also allows the composed object to be declared as a `const` object.

### 5.5.2 Composition and reference objects: reference memberinitializers

Apart from using member initializers to initialize composed objects (be they `const` objects or not), there is another situation where member initializers must be used. Consider the following situation.

A program uses an object of the class `Configfile`, defined in `main()` to access the information in a configuration file. The configuration file contains parameters of the program which may be set by changing the values in the configuration file, rather than by supplying command line arguments.

Assume that another object that is used in the function `main()` is an object of the class `Process`, doing 'all the work'. What possibilities do we have to tell the object of the class `Process` that an object of the class `Configfile` exists?

- The objects could have been declared as *global* objects. This *is* a possibility, but not a very good one, since all the advantages of local objects are lost.
- The `Configfile` object may be passed to the `Process` object at construction time. Passing an object in a blunt way (i.e., by value) might not be a very good idea, since the object must be copied into the `Configfile` parameter, and then a data member of the `Process` class can be used to make the `Configfile` object accessible throughout the `Process` class. This might involve yet another object-copying task, as in the following situation:

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

- The copy-instructions can be avoided by using *pointers* to the `Configfile` objects, as in:

```
Process::Process(Configfile *conf)  // a pointer to an external object
{
    conf_ptr = conf;                // the conf_ptr is a Configfile *
    ...
}
```

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

- Rather than using value or pointer parameters, the Configfile parameter could be defined as a *reference parameter* to the Process constructor. Next, we can define a Config reference data member in the class Process. Using the reference variable effectively uses a pointer, disguised as a variable.

However, the following construction will *not* result in the correct initialization of the Configfile &conf_ref reference data member:

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

The statement conf_ref = conf fails, because the compiler won't see this as an initialization, but considers this an assignment of one Configfile object (i.e., conf), to another (conf_ref). It does so, because that's the normal interpretation: an assignment to a reference variable is actually an assignment to the variable the reference variable refers to. But to what variable does conf_ref refer? To no variable, since we haven't initialized conf_ref. After all, the whole purpose of the statement conf_ref = conf was to initialize conf_ref....

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

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

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

```
Process::Process(int &ir)
:
    int_ref(ir)
{
    ...
}
```

would have been called for.

## 5.6 Friend functions and friend classes

As we have seen in the previous sections, `private` data or function members are normally only accessible by the code which is part of the corresponding class. However, situations may arise in which it is desirable to allow the explicit access to `private` members of one class to one or more other classless functions or member functions of classes.

E.g., consider the following code example (all functions are `inline` for purposes of brevity):

```
class A                         // class A: just stores an
{                               // int value via the constructor
    public:                     // and can retrieve it via
        A(int v)                // getval
            { value = v; }
        int getval()
            { return (value); }

    private:
        int value;
};

void decrement(A &a)            // function decrement: tries
{                               // to alter A's private data
    a.value--;
}

class B                         // class B: tries to touch
{                               // A's private parts
    public:
        void touch(A &a)
            { a.value++; }
};
```

This code will not compile, since the classless function `decrement()` and the function `touch()` of the class `B` attempt to access a `private` datamember of `A`.

We can explicitly allow `decrement()` to access `A`'s data, and we can explicitly allow the class `B` to access these data. To accomplish this, the offending classless function `decrement()` and the class `B` are declared to be friends of `A`:

```
class A
{
    public:
        friend class B;             // B's my buddy, I trust him

        friend void decrement(A     // decrement() is also a good pal
            &what);
        ...
};
```

Concerning friendship between classes, we remark the following:

- Friendship is not mutual by default. This means that once `B` is declared as a friend of `A`, this does not give `A` the right to access `B`'s private members.
- Friendship, when applied to program design, is an *escape mechanism* which circumvents the principle of data hiding. Using friend classes should therefore be *minimized* to those cases where it is absolutely essential.
- If friends are used, realize that the implementation of classes or functions that are friends to other classes become implementation dependent on these classes. In the above example: once the internal organization of the data of the class `A` changes, all its friends must be recompiled (and possibly modified) as well.
- As a rule of thumb: *don't* use friend functions or classes.

Having thus issued some warnings against the use of friends, we'll leave our discussion of friends for the time being. However, in section 14 we'll continue the discussion, having covered, by that time, the topic of operator overloading.

## 5.7 Header file organization with classes

In section 2.5.11 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 in the `include`-directives and to think about the header files which are needed in a particular source file.

The second alternative has the advantage of economy for the program developer: the header file of the class accumulates header files, so it tends to become more and more generally useful. It has the disadvantage that the compiler will often have to read header files which aren't actually used by the function defined in the source file.

With computers running faster and faster we think the second alternative is to be preferred over the first alternative. So, 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 (comparable to, e.g., the `sys` subdirectory). If `class MyClass` is developed as part of some larger project, create a subdirectory (or subdirectory link) of one of the `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 <myproject/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. The `File` class has a member `gets(String &destination)`, which reads a line from a file, and stores the line in the `String` object passed to the `gets()` member function as reference, while the class `String` has a member function `getLine(File &file)`, which reads one line from the `File` object which is passed to the `getLine()` member function as a reference. The (partial) header file for the `class String` is then:

```
#ifndef _String_h_
#define _String_h_

#include <project/file.h>   // to know about a File

class String
{
    public:
        void getLine(File &file);
    ...
};
#endif
```

However, a similar setup is required for the class `File`:

```
#ifndef _File_h_
#define _File_h_

#include <project/string.h>   // to know about a String

class File
{
    public:
```

```
            void gets(String &string);
        ...
};
#endif
```

Now we have created a problem. The compiler, trying to compile `File::gets()` proceeds as follows:

- The header file `project/string.h` is opened to be read
- `_String_h_` is defined
- The header file `project/file.h` is opened to be read
- `_File_h_` is defined
- The header file `project/string.h` is opened to be read
- `_String_h_` has been defined, so `project/string.h` is skipped
- The definition of the `class File` is parsed.
- In the class definition contains a reference to a `String` object
- As the `class String` hasn't been parsed yet, a `String` is an undefined type, and the compiler quits with an error.

The solution for this problem is to use a *forward class reference* before the class definition, and to include the corresponding class header file after the class definition. So we get:

```
#ifndef _String_h_
#define _String_h_

class File;                 // forward reference

class String
{
    public:
        void getLine(File &file);
    ...
};

#include <project/file.h>   // to know about a File

#endif
```

However, a similar setup is required for the class `File`:

```
#ifndef _File_h_
```

```
#define _File_h_

class String;               // forward reference

class File
{
    public:
        void gets(String &string);
    ...
};

#include <project/string.h>   // to know about a String

#endif
```

This works well in all situations where either references or pointers to another class are involved. But it doesn't work with composition. Assume the class `File` has a *composed* data member of the class `String`. In that case, the class definition of the class `File` *must* include the header file of the class `String` before the class definition itself, because otherwise the compiler can't tell how big a `File` object will be, as it doesn't know the size of a `String` object once the definition of the `File` class is completed.

In cases where classes contain composed objects (or are derived from other classes, see chapter 15) the header files of the classes of the composed objects must have been read *before* the class definition itself. In such a case the `class File` might be defined as follows:

```
#ifndef _File_h_
#define _File_h_

#include <project/string.h>   // to know about a String

class File
{
    public:
        void gets(String &string);
    ...
    private:
        String              // composition !
            line;
};
#endif
```

Note that the class `String` can't have a `File` object as a composed member: such a situation would result again in an undefined class while compiling the sources of these classes.

All other required header files are either related to classes that are used only within the source files themselves (without being part of the current class definition), or they are related to classless functions (like `memcpy()`). All headers that are not required by the compiler to parse the current class definition can be mentioned below the class definition.

To summarize, a `class` header file should be organized as follows:

- Everything is contained within the block defined by the standard `ifndef` and `endif` directives.
- Header files of classes of objects that are either composed or inherited (see chapter 15) are mentioned first.
- The classes of objects appearing only as references or as pointers in the class definition are specified as forward references.
- Next comes the class definition itself.
- Following the class definition the header files of all classes given as forward references are included.
- Finally, all other header files that are required in the source files of the class are included.

An example of such an header file is:

```
#ifndef _File_h_
#define _File_h_

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

class String;           // forward reference

class File              // class definition
{
    public:
        void gets(String &string);
    ...
    private:
        ifstream
            instream;
};
                        // for the class String
#include <project/string.h>

                        // for remaining software
#include <memory.h>
#include <sys/stat.h>

#endif
```

## 5.8 Nesting Classes

Classes can be defined inside other classes. Classes that are defined inside other classes are called *nested classes.*

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 visibility 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 15), if it is nested in the `private` section, it is only visible for the members of the surrounding class.

The surrounding class has no privileges with respect to the nested class. So, the nested class still has full control over the accessibility of its members by the surrounding class.

For example, consider the following class definition:

```
class Surround
{
    public:
        class FirstWithin
        {
            public:
                FirstWithin();
                int getVar() const
                {
                    return (variable);
                }
            private:
                int
                    variable;
        };
    private:
        class SecondWithin
        {
            public:
                SecondWithin();
                int getVar() const
                {
                    return (variable);
                }
            private:
                int
                    variable;
        };
        // other private members of Surround
};
```

In this definition access to the members is defined as follows:

- The class `FirstWithin` is visible both outside and inside `Surround`. The class `FirstWithin` has therefore global scope.
- The constructor `FirstWithin()` and the member function `getVar()` of the class `FirstWithin` are also globally visible.

- The `int variable` datamember is only visible for the members of the class `FirstWithin`. Neither the members of `Surround` nor the members of `SecondWithin` can access the `variable` of the class `FirstWithin` directly.
- The class `SecondWithin` is visible only inside `Surround`. The public members of the class `SecondWithin` can also be used by the members of the class `FirstWithin`, as nested classes can be considered members of their surrounding class.
- The constructor `SecondWithin()` and the member function `getVar()` of the class `SecondWithin` can also only be reached by the members of `Surround` (and by the members of its nested classes).
- The `int variable` datamember of the class `SecondWithin` is only visible for the members of the class `SecondWithin`. Neither the members of `Surround` nor the members of `FirstWithin` can access the `variable` of the class `SecondWithin` directly.

If the surrounding class should have access rights to the private members of its nested classes or if nested classes should have access rights to the private members of the surrounding class, the classes can be defined as `friend` classes (see section 5.8.3).

The nested classes can be considered members of the surrounding class, but the members of nested classes are *not* members of the surrounding class. So, a member of the class `Surround` may not access `FirstWithin::getVar()` directly. This is understandable considering the fact that a `Surround` object is not also a `FirstWithin` or `SecondWithin` object. The nested classes are only available as typenames. They do not imply containment as objects by the surrounding class. If a member of the surrounding class should use a (non-static) member of a nested class then a pointer to a nested class object or a nested class datamember must be defined in the surrounding class, which can thereupon be used by the members of the surrounding class to access members of the nested class.

For example, in the following class definition there is a surrounding class `Outer` and a nested class `Inner`. The class `Outer` contains a member function `caller()` which uses the `inner` object that is composed in `Outer` to call the `infunction()` member function of `Inner`:

```
class Outer
{
    public:
        void caller()
        {
            inner.infunction();
        }
    private:
        class Inner
        {
            public:
                void infunction();
        };
        Inner
            inner;
};
```

Also note that the function `Inner::infunction()` can be called as part of the inline definition of `Outer::caller()`, even though the definition of the class `Inner` is yet to be seen by the compiler.

Inline functions can be defined as if they were functions that were defined outside of the class definition: if the function `Outer::caller()` would have been defined outside of the class `Outer`, the full class definition (including the definition of the class `Inner` would have been available to the compiler. In that situation the function is perfectly compilable. Inline functions can be compiled accordingly and there is, e.g., no need to define a special private section in `Outer` in which the class `Inner` is defined before defining the inline function `caller()`.

### 5.8.1 Defining nested class members

Member functions of nested classes may be defined as inline functions. However, they can also be defined outside of their surrounding class. Consider the constructor of the class `FirstWithin` in the example of the previous section. The constructor `FirstWithin()` is defined in the class `FirstWithin`, which is, in turn, defined within the class `Surround`. Consequently, the class scopes of the two classes must be used to define the constructor. E.g.,

```
Surround::FirstWithin::FirstWithin()
{
    variable = 0;
}
```

Static (data) members can be defined accordingly. If the class `FirstWithin` would have a `static unsigned` datamember `epoch`, it could be initialized as follows:

```
Surround::FirstWithin::epoch = 1970;
```

Furthermore, both class scopes are needed to refer to public static members in code outside of the surrounding class:

```
void showEpoch()
{
    cout << Surround::FirstWithin::epoch = 1970;
}
```

Of course, inside the members of the class `Surround` only the `FirstWithin::` scope needs to be mentioned, and 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 directy 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
```

### 5.8.2 Declaring nested classes

Nested classes may be declared before they are actually defined in a surrounding class. Such forward declarations are required if a class contains multiple nested classes, and the nested classes contain pointers to objects of the other nested classes.

For example, the following class `Outer` contains two nested classes `Inner1` and `Inner2`. The class `Inner1` contains a pointer to `Inner2` objects, and `Inner2` contains a pointer to `Inner1` objects. Such cross references require forward declarations:

```
class Outer
{
    ...
    private:
        class Inner2;       // forward declaration

        class Inner1
        {
            ...
            private:
                Inner2
                    *pi2;   // points to Inner2 objects
        };
        class Inner2
        {
            ...
            private:
                Inner1
                    *pi1;   // points to Inner1 objects
        };
        ...
};
```

### 5.8.3 Access to private members in nested classes

In order to allow nested classes to access the private members of the surrounding class or to access the private members of other nested classes or to allow the surrounding class to access the private members of nested classes, the `friend` keyword must be used. Consider the following situation, in which a class `Surround` has two nested classes `FirstWithin` and `SecondWithin`, while each class has a static data member `int variable`:

```
class Surround
{
    public:
        class FirstWithin
```

```
        {
            public:
                int getValue();
            private:
                static int
                    variable;
        };
        int getValue();
    private:
        class SecondWithin
        {
            public:
                int getValue();
            private:
                static int
                    variable;
        };
        static int
            variable;
};
```

If the class `Surround` should be able to access the private members of `FirstWithin` and `SecondWithin`, these latter two classes must declare `Surround` to be their friend. The function `Surround::getValue()` can thereupon access the private members of the nested classes. For example (note the `friend` declarations in the two nested classes):

```
class Surround
{
    public:
        class FirstWithin
        {
            friend class Surround;
            public:
                int getValue();
            private:
                static int
                    variable;
        };
        int getValue()
        {
            FirstWithin::variable = SecondWithin::variable;
            return (variable);
        }
    private:
        class SecondWithin
        {
            friend class Surround;
            public:
                int getValue();
            private:
```

```
                static int
                    variable;
        };
        static int
            variable;
};
```

Now, in order to allow the nested classes to access the private members of the surrounding class, the class `Surround` must declare the nested classes as friends. The `friend` keyword may only be used when the class that is to become a friend is already known as a class by the compiler, so either a *forward* declaration of the nested classes is required, which is followed by the friend declaration, or the friend declaration follows the definition of the nested classes. The forward declaration followed by the friend declaration looks like this:

```
class Surround
{
    class FirstWithin;
    class SecondWithin;
    friend class FirstWithin;
    friend class SecondWithin;

    public:
        class FirstWithin

        ... (etc)
```

Alternatively, the friend declaration may follow the definition of the classes. Note that a class can be declared a friend following its definition, while the inline code in the definition already uses the fact that it will be declared a friend of the outer class. Also note that the inline code of the nested class uses members of the surrounding class which have not yet been seen by the compiler. Finally note that the variable `variable` that is defined in the class `Surround` is accessed in the nested classes as `Surround::variable`:

```
class Surround
{
    public:
        class FirstWithin
        {
            friend class Surround;
            public:
                int getValue()
                {
                    Surround::variable = 4;
                    return (variable);
                }
            private:
                static int
                    variable;
```

```
        };
        friend class FirstWithin;

        int getValue()
        {
            FirstWithin::variable = SecondWithin::variable;
            return (variable);
        }
    private:
        class SecondWithin
        {
            friend class Surround;
            public:
                int getValue()
                {
                    Surround::variable = 40;
                    return (variable);
                }
            private:
                static int
                    variable;
        };
        friend class SecondWithin;

        static int
            variable;
};
```

Finally, we want to allow the nested classes to access each other's private members. Again this requires some friend declarations. In order to allow `FirstWithin` to access `SecondWithin`'s private members nothing but a friend declaration in `SecondWithin` is required. However, to allow `SecondWithin` to access the private members of `FirstWithin` the `friend class SecondWithin` declaration cannot be plainly given in the class `FirstWithin`, as the definition of `SecondWithin` has not yet been given. A forward declaration of `SecondWithin` is required, and this forward declaration must be given in the class `Surround`, rather than in the class `FirstWithin`. Clearly, the forward declaration `class SecondWithin` in the class `FirstWithin` itself makes no sense, as this would refer to an external (global) class `FirstWithin`. But the attempt to provide the forward declaration of the nested class `SecondWithin` inside `FirstWithin` as `class Surround::SecondWithin` also fails miserably, with the compiler issuing a message like

```
'Surround' does not have a nested type named 'SecondWithin'
```

The right procedure to follow here is to declare the class `SecondWithin` in the class `Surround`, before the class `FirstWithin` is defined. Using this procedure, the friend declaration of `SecondWithin` is accepted inside the definition of `FirstWithin`. The following class definition allows full access of the private members of all classes by all other classes:

```
class Surround
{
    class SecondWithin;
```

```
    public:
        class FirstWithin
        {
            friend class Surround;
            friend class SecondWithin;
            public:
                int getValue()
                {
                    Surround::variable = SecondWithin::variable;
                    return (variable);
                }
            private:
                static int
                    variable;
        };
        friend class FirstWithin;

        int getValue()
        {
            FirstWithin::variable = SecondWithin::variable;
            return (variable);
        }
    private:
        class SecondWithin
        {
            friend class Surround;
            friend class FirstWithin;
            public:
                int getValue()
                {
                    Surround::variable = FirstWithin::variable;
                    return (variable);
                }
            private:
                static int
                    variable;
        };
        friend class SecondWithin;

        static int
            variable;
};
```

### 5.8.4 Nesting enumerations

Enumerations may also be nested in classes. For example, a class `DataStructure` may be traversed in a forward or backward direction. Such a class can define an enumerator `Traversal` having the values `forward` and `backward`. Furthermore, a member function `setTraversal()` can be defined requiring either of the two enumeration values. The class can be defined as follows:

```
class DataStructure
{
    public:
        enum Traversal
        {
            forward,
            backward
        };
        setTraversal(Traversal mode);
        ...
    private:
        Traversal
            mode;
        ...
};
```

Within the class DataStructure the values of the Traversal enumeration can be used directly. For example:

```
void DataStructure::setTraversal(Traversal modeArg)
{
    mode = modeArg;
    switch (mode)
    {
        forward:
            ....
        break;

        backward:
            ....
        break;
    }
}
```

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 enough. Only if a variable of the enumeration type is required the name of the enumeration type is needed, as illustrated by the following piece of code:

```
void fun()
{
    DataStructure::Traversal                // enum typename required
        localMode = DataStructure::forward; // enum typename not required

    DataStructure
        ds;
                                            // enum typename not required
```

```
        ds.setTraversal(DataStructure::backward);
    }
```

Again, if `DataStructure` would define a nested class `Nested` in which the enumeration `Traversal` would have been defined, the two class scopes would have been required. In that case the former example would have to be coded as follows:

```
    void fun()
    {
        DataStructure::Nested::Traversal
            localMode = DataStructure::Nested::forward;

        DataStructure
            ds;

        ds.setTraversal(DataStructure::Nested::backward);
    }
```

# Chapter 6

# 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 5.3.3) which can be used instead of the explicit checks for `NULL`-returns.

The relationship between `free()` and `delete` is analogous: `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.

For the function `strdup()` a comparable function `char *strdupnew(char const *str)` could be developed as follows:

```
char *strdupnew(char const *str)
```

```
{
    return (strcpy(new char [strlen(str) + 1], str));
}
```

Similar functions could be developed for comparable `malloc()`-based `str...()` and other functions.

In this chapter we discuss the following topics:

- the assignment operator (and operator overloading in general),
- the `this` pointer,
- the copy constructor.

## 6.1 Classes with pointer data members

In this section we shall again use the class `Person` as example:

```
class Person
{
    public:
        // constructors and destructor
        Person();
        Person(char const *n, char const *a,
               char const *p);
        ~Person();

        // interface functions
        void setname(char const *n);
        void setaddress(char const *a);
        void setphone(char const *p);

        char const *getname(void) const;
        char const *getaddress(void) const;
        char const *getphone(void) const;

    private:
        // data fields
        char *name;
        char *address;
        char *phone;
};
```

In this class the destructor is necessary to prevent that memory, once allocated for the fields `name`, `address` and `phone`, becomes unreachable when an object ceases to exist. In the following example a `Person` object is created, after which the data fields are printed. After this the `main()` function stops, which leads to the deallocation of memory. The destructor of the class is also shown for illustration purposes.

Note that in this example an object of the class `Person` is also created and destroyed using a pointer variable; using the operators `new` and `delete`.

```
Person::~Person()
{
    delete name;
    delete address;
    delete phone;
}

int main()
{
    Person
        kk("Karel", "Rietveldlaan",
            "050 542 6044"),
        *bill = new Person("Bill Clinton",
               "White House",
               "09-1-202-142-3045");

    printf("%s, %s, %s\n"
           "%s, %s, %s\n",
        kk.getname(), kk.getaddress(), kk.getphone(),
        bill->getname(), bill->getaddress(), bill->getphone());

    delete bill;

    return (0);
}
```

The memory occupied by the object `kk` is released automatically when `main()` terminates: the **C++** compiler makes sure that the destructor is called. Note, however, that the object pointed to by `bill` is handled differently. The variable `bill` is a pointer; and a pointer variable is, even in **C++**, in itself no `Person`. Therefore, before `main()` terminates, the memory occupied by the object pointed to by `bill` must be *explicitly* released; hence the statement `delete bill`. The operator `delete` will make sure that the destructor is called, thereby releasing the three strings of the object.

## 6.2 The assignment operator

Variables which are `structs` or `classes` can be directly assigned in **C++** in the same way that `structs` can be assigned in **C**. The default action of such an assignment is a straight bytewise copy from one compound variable to another.

Let us now consider the consequences of this default action in a program statement such as the following:

```
void printperson(Person const &p)
{
    Person
        tmp;

    tmp = p;
    printf("Name:     %s\n"
            "Address:  %s\n"
            "Phone:    %s\n",
        tmp.getname(), tmp.getaddress(), tmp.getphone());
}
```

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 6.1.

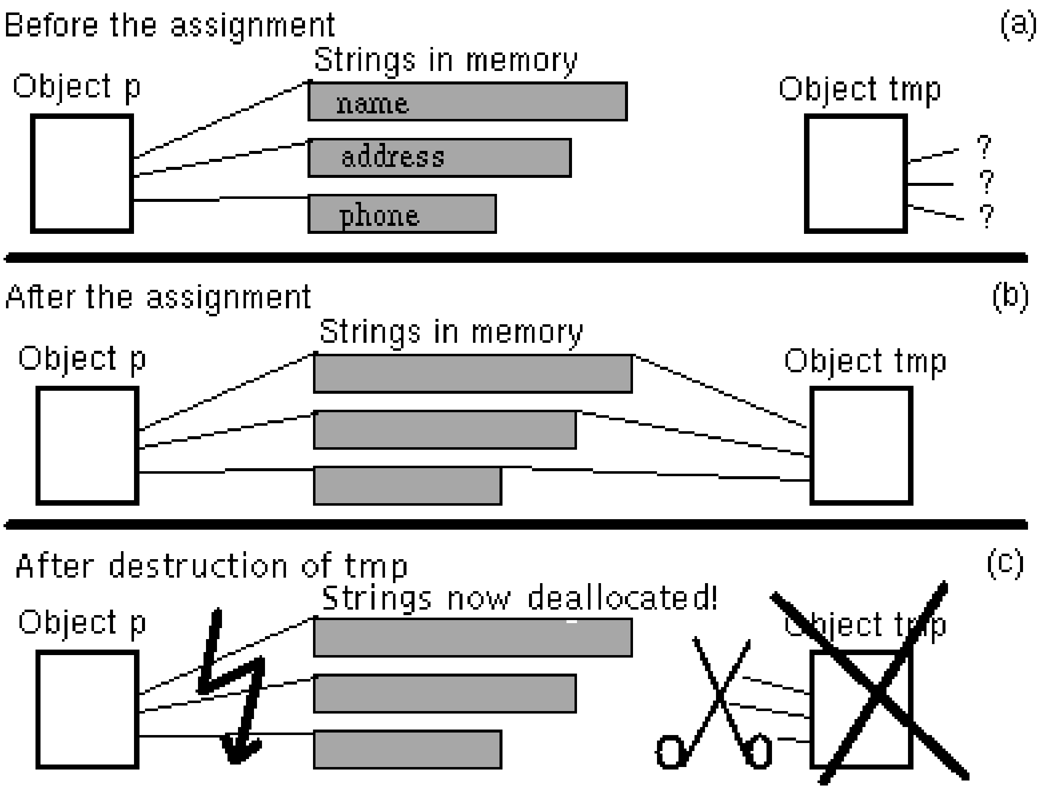
Having executed `printperson()`, the object which was referenced by `p` now contain pointers to deallocated memory.

This action is undoubtedly not a desired effect of a function like the above. The deallocated memory will likely become occupied during subsequent allocations: the pointer members of `p` have effectively become *wild pointers*, as they don't point to allocated memory anymore.

In general it can be concluded that *every class containing pointer data members is a potential candidate for trouble*. It is of course possible to prevent such troubles, as will be discussed in the next section.

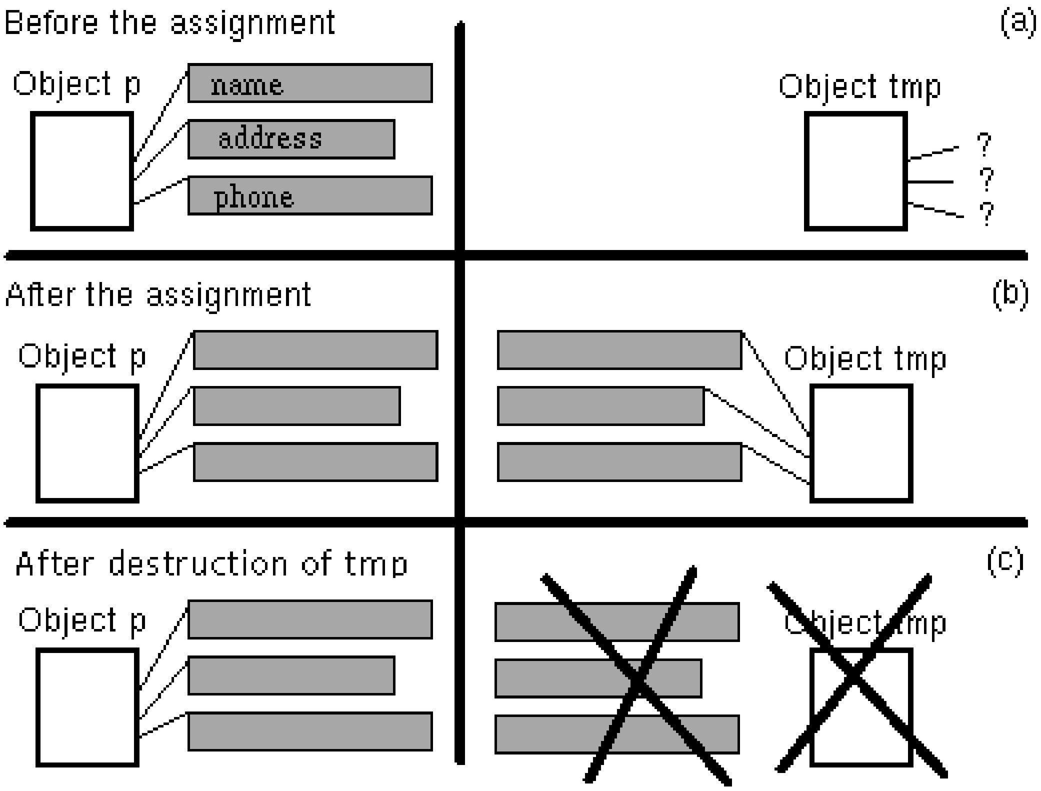
### 6.2.1 Overloading the assignment operator

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.




\

Figure 6.1: Private data and public interface functions of the class Person, using bytewise assignment




\

Figure 6.2: Private data and public interface functions of the class Person, using the 'correct' assignment.

The 'right' way to duplicate a `Person` object is illustrated in figure 6.2.

There is a number of solutions for the above wish. One solution consists of the definition of a special function to handle assignments of objects of the class `Person`. The purpose of this function would be to create a copy of an object, but one with its own `name`, `address` and `phone` strings. Such a member function might be:

```
void Person::assign(Person const &other)
{
    // delete our own previously used memory
    delete name;
    delete address;
    delete phone;

    // now copy the other Person's data
    name = strdupnew(other.name);
    address = strdupnew(other.address);
    phone = strdupnew(other.phone);
}
```

Using this tool we could rewrite the offending function `printperson()`:

```
void printperson(Person const &p)
{
    Person
        tmp;

    // make tmp a copy of p, but with its own allocated
    // strings
    tmp.assign(p);

    printf("Name:     %s\n"
           "Address:  %s\n"
           "Phone:    %s\n",
        tmp.getname(), tmp.getaddress(), tmp.getphone());

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

In itself this solution is valid, although it is a purely symptomatic solution. This solution requires that the programmer uses a specific member function instead of the operator `=`. The problem, however, remains if this rule is not strictly adhered to. Experience learns that *errare humanum est*: a solution which doesn't enforce exceptions is therefore preferable.

The problem of the assignment operator is solved by means of *operator overloading*: the syntactic possibility **C++** offers to redefine the actions of an operator in a given context. Operator overloading was mentioned earlier, when the operators << and >> were redefined for the usage with streams as `cin`, `cout` and `cerr` (see section 3.1.2).

Overloading the assignment operator is probably the most common form of operator overloading. However, a word of warning is appropriate: the fact that **C++** allows operator overloading does not mean that this feature should be used at all times. A few rules are:

- Operator overloading should be used in situations where an operator has a defined action, but when this action is not desired as it has negative side effects. A typical example is the above assignment operator in the context of the class `Person`.
- Operator overloading can be used in situations where the usage of the operator is common and when no ambiguity in the meaning of the operator is introduced by redefining it. An example may be the redefinition of the operator `+` for a class which represents a complex number. The meaning of a `+` between two complex numbers is quite clear and unambiguous.
- In all other cases it is preferable to define a member function, instead of redefining an operator.

Using these rules, operator overloading is minimized which helps keep source files readable. An operator simply does what it is designed to do. Therefore, in our vision, the operators insertion (<<) and extraction (>>) operators in the context of streams are unfortunate: the stream operations do not have anything in common with the bitwise shift operations.

### The function 'operator=()'

To achieve operator overloading in the context of a class, the class is simply expanded with a `public` function stating the particular operator. A corresponding function, the implementation of the overloaded operator, is thereupon defined.

For example, to overload the addition operator `+`, a function `operator+()` must be defined. The function name consists of two parts: the keyword `operator`, followed by the operator itself.

In our case we define a new function `operator=()` to redefine the actions of the assignment operator. A possible extension to the class `Person` could therefore be:

```
// new declaration of the class
class Person
{
    public:
        ...
        void operator=(Person const &other);
        ...
    private:
        ...
};

// definition of the function operator=()
void Person::operator=(Person const &other)
{
    // deallocate old data
    delete name;
    delete address;
```

```
        delete phone;

        // make duplicates of other's data
        name = strdupnew(other.name);
        address = strdupnew(other.address);
        phone = strdupnew(other.phone);
    }
```

The function `operator=()` presented here is the first version of the overloaded assignment operator. We shall present better and less bug-prone versions shortly.

The actions of this member function are similar to those of the previously proposed function `assign()`, but now its *name* makes sure that this function is also activated when the assignment operator = is used. There are actually two ways to call this function, as illustrated below:

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

    // first possibility
    copy = pers;

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

It is obvious that the second possibility, in which `operator=()` is explicitly stated, is not used often. However, the code fragment *does* illustrate the two ways of calling the same function.

## 6.3 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[1]

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` or `private`). Therefore, it is as if in each member function of the class `Person` would contain the following declaration:

```
    extern Person *this;
```

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

[1]Note that '`this`' is not available in the not yet discussed `static` member functions.

```
// alternative 1: implicit usage of this
void Person::setname(char const *n)
{
    delete name;
    name = strdupnew(n);
}

// alternative 2: explicit usage of this
void Person::setname(char const *n)
{
    delete this->name;
    this->name = strdupnew(n);
}
```

Explicit usage of the `this` pointer is not used very frequently. However, there exist a number of situations where the `this` pointer is really needed.

### 6.3.1 Preventing self-destruction with this

As we have seen, the operator `=` can be redefined for the class `Person` in such a way that two objects of the class can be assigned, leading to 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 released, but this also leads to the release of the strings of the right-hand side variable, which we call *self-destruction*. An example of this situation is illustrated below:

```
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,
    two,
    *pp;

pp = &one;
...
*pp = two;
```

```
...
one = *pp;
```

The problem of the 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=()` then becomes:

```
void Person::operator=(Person const &other)
{
    // only take action if address of current object
    // (this) is NOT equal to address of other
    // object(&other):

    if (this != &other)
    {
        delete name;
        delete address;
        delete phone;

        name = strdupnew(other.name);
        address = strdupnew(other.address);
        phone = strdupnew(other.phone);
    }
}
```

This is the second version of the overloaded assignment function. One, yet better version remains to be discussed.

As a subtlety, note the usage of the *address operator* '`&`' in the statement

```
if (this != &other)
```

The variable `this` is a pointer to the 'current' object, while `other` is a reference; which is an 'alias' to an actual `Person` object. The address of the other object is therefore `&other`, while the address of the current object is `this`.

### 6.3.2 Associativity of operators and this

According to **C++**'s syntax, the associativity of the assignment operator is to the right-hand side. I.e., in statements like:

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

the expression `b = c` is evaluated first, and the result is assigned to `a`.

The implementation of the overloaded assignment operator so far does not permit such constructions, as an assignment using the member function returns nothing (`void`). We can therefore conclude that the previous implementation does circumvent an allocation problem, but is syntactically not quite right.

The syntactical 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`; and the class `Person` contains no member functions with the prototype `operator=(void)`.

This problem can also be remedied using the `this` pointer. The overloaded assignment function expects as its argument a reference to a `Person` object. It can also *return* a reference to such an object. This reference can then be used as an argument for a nested assignment.

It is customary to let the overloaded assignment return a reference to the current object (i.e., `*this`), as a `const` reference: the receiver is not supposed to alter the `*this` object.

The (final) version of the overloaded assignment operator for the class `Person` thus becomes:

```
// declaration in the class
class Person
{
    public:
        ...
        Person const &operator=(Person const &other)
        ...
};

// definition of the function
Person const &Person::operator=(Person const &other)
{
    // only take action when no auto-assignment occurs
    if (this != &other)
    {
        // deallocate own data
        delete address;
        delete name;
        delete phone;

        // duplicate other's data
        address = strdupnew(other.address);
        name = strdupnew(other.name);
        phone = strdupnew(other.phone);
    }
```

```
        // return current object, compiler will make sure
        // that a const reference is returned
        return (*this);
    }
```

## 6.4 The copy constructor: Initialization vs. Assignment

In the following sections we shall take a closer look at another usage of the operator =. For this, we shall use a class `String`. This class is meant to handle allocated strings, and its interface is as follows:

```
class String
{
    public:
        // constructors, destructor
        String();
        String(char const *s);
        ~String();

        // overloaded assignment
        String const &operator=(String const &other);

        // interface functions
        void set(char const *data);
        char const *get(void);

    private:
        // one data field: ptr to allocated string
        char *str;
};
```

Concerning this interface we remark the following:

- The class contains a pointer `char *str`, possibly pointing to allocated memory. Consequently, the class needs a constructor and a destructor.

  A typical action of the constructor would be to set the `str` pointer to 0. A typical action of the destructor would be to release the allocated memory.

- For the same reason the class has an overloaded assignment operator. The code of this function would look like:

  ```
  String const &String::operator=(String const &other)
  {
      if (this != &other)
      {
  ```

```
                delete str;
                str = strdupnew(other.str);
            }
            return (*this);
        }
```

- The class has, besides a default constructor, a constructor which expects one string argument. Typically this argument would be used to set the string to a given value, as in:

```
        String
            a("Hello World!\n");
```

- The only interface functions are to set the string part of the object and to retrieve it.

Now let's consider the following code fragment. The statement references are discussed following the example:

```
String
    a("Hello World\n"),             // see (1)
    b,                              // see (2)
    c = a;                          // see (3)

int main()
{
    b = c;                          // see (4)
    return (0);
}
```

- Statement 1: this statement shows an initialization. The object `a` is initialized with a string "Hello World". This construction of the object `a` therefore uses the constructor which expects one string argument.

  It should be noted here that this form is identical to

```
        String
            a = "Hello World\n";
```

  Even though this piece of code uses the operator `=`, this is no assignment: rather, it is an *initialization*, and hence, it's done at *construction time* by a constructor of the class `String`.

- Statement 2: here a second `String` 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 `c` 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 `a`.

  This form of initializations has not yet been discussed. As we can rewrite this statement in the form

```
String
    c(a);
```

it suggests that a constructor is called, with as argument a (reference to a) `String` object. Such constructors are quite common in **C++** and are called *copy constructors.* More properties of these constructors are discussed below.

- Statement 4: here one object is assigned to another. No object is *created* in this statement. Hence, this is just an assignment, using the overloaded assignment operator.

The simple rule emanating from these examples is that *whenever an object is created, a constructor is needed.* All constructors have the following characteristics:

- Constructors have no return values.
- Constructors are defined in functions having the same names as the class to which they belong.
- The argument list of constructors can be deduced from the code. The argument is either present between parentheses or following a `=`.

Therefore, we conclude that, given the above statement (3), the class `String` must be rewritten to define a **copy constructor**:

```
// class definition
class String
{
    public:
        ...
        String(String const &other);
        ...
};

// constructor definition
String::String(String const &other)
{
    str = strdupnew(other.str);
}
```

The actions of copy constructors are comparable to those of the overloaded assignment operators: an object is *duplicated*, so that it contains its own allocated data. The copy constructor function, however, is simpler in the following respect:

- A copy constructor doesn't need to deallocate previously allocated memory: since the object in question has just been created, it cannot already have its own allocated data.
- A copy constructor never needs to check whether auto-duplication occurs. No variable can be initialized with itself.

Besides the above mentioned quite obvious usage of the copy constructor, the copy constructor has other important tasks. All of these tasks are related to the fact that the copy constructor is always called when an object is created and initialized with another object of its class. The copy constructor is called even when this new object is a hidden or temporary variable.

- When a function takes an object as argument, instead of, e.g., a pointer or a reference, **C++** calls the copy constructor 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 func(String s)         // no pointer, no reference
{                           // but the String itself
    puts(s.get());
}

int main()
{
    String
        hi("hello world");

    func(hi);
    return (0);
}
```

  In this code fragment `hi` itself is not passed as an argument, but instead a temporary(stack) variable is created using the copy constructor. This temporary variable is known within `func()` as `s`. Note that if `func()` would have been defined using a reference argument, 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.

  This situation occurs when, e.g., a function returns keyboard input in a `String` format:

```
String getline()
{
    char
        buf [100];          // buffer for kbd input

    gets(buf);              // read buffer

    String
        ret = buf;          // convert to String

    return(ret);            // and return it
}
```

  A hidden `String` object is here initialized with the return value `ret` (using the copy constructor) and is returned by the function. The local variable `ret` itself ceases to exist when `getline()` terminates.

To demonstrate that copy constructors are not called in all situations, consider the following. We could rewrite the above function `getline()` to the following form:

```
String getline()
{
    char
        buf [100];          // buffer for kbd input

    gets(buf);              // read buffer
    return (buf);           // and return it
}
```

This code fragment is quite valid, even though the return value `char *` doesn't match the prototype `String`. In this situation, C++ will try to convert the `char *` to a `String`. It can do so given a constructor expecting a `char *` argument. This means that the copy constructor is *not* used in this version of `getline()`. Instead, the constructor expecting a `char *` argument is used.

Contrary to the situation we encountered with the default constructor, the default copy constructor remains available once a constructor (*any constructor*) is defined explicitly. The copy constructor can be redefined, but it will not disappear once another constructor is defined.

### 6.4.1 Similarities between the copy constructor and operator=()

The similarities between on one hand the copy constructor and on the other hand the overloaded assignment operator are reinvestigated in this section. We present here two primitive functions which often occur in our code, and which we think are quite useful. Note the following features of copy constructors, overloaded assignment operators, and destructors:

- The *duplication of (private) data* occurs (1) in the copy constructor and (2) in the overloaded assignment function.
- The *deallocation of used memory* occurs (1) in the overloaded assignment function and (2) in the destructor.

The two above actions (duplication and deallocation) can be coded in two private functions, say `copy()` and `destroy()`, which are used in the overloaded assignment operator, the copy constructor, and the destructor. When we apply this method to the class `Person`, we can rewrite the code 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 deallocate the memory of the current object *unconditionally*. Hence these functions implement 'primitive' functionality:

```
// class definition, only relevant functions are shown here
class Person
{
    public:
```

```
        // constructors, destructor
        Person(Person const &other);
        ~Person();

        // overloaded assignment
        Person const &operator=(Person const &other);
    private:
        // data fields
        char
            *name,
            *address,
            *phone;

        // the two primitives
        void copy(Person const &other);
        void destroy(void);
};
```

Next, we present the implementations of the functions `copy()` and `destroy()`:

```
// copy(): unconditionally copy other object's data
void Person::copy(Person const &other)
{
    name = strdupnew(other.name);
    address = strdupnew(other.address);
    phone = strdupnew(other.phone);
}

// destroy(): unconditionally deallocate data
void Person::destroy ()
{
    delete name;
    delete address;
    delete phone;
}
```

Finally the three `public` functions in which other object's memory is copied or in which memory is deallocated are rewritten:

```
// copy constructor
Person::Person (Person const &other)
{
    // unconditionally copy other's data
    copy(other);
}

// destructor
```

```
Person::~Person()
{
    // unconditionally deallocate
    destroy();
}

// overloaded assignment
Person const &Person::operator=(Person const &other)
{
    // only take action if no auto-assignment
    if (this != &other)
    {
        destroy();
        copy(other);
    }
    // return (reference to) current object for
    // chain-assignments
    return (*this);
}
```

What we like about this approach is that the destructor, copy constructor and overloaded assignment functions are 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()`.

## 6.5 Conclusion

Two important extensions to classes have been discussed in this chapter: the overloaded assignment operator and the copy constructor. As we have seen, classes with pointer data which address allocated memory are potential sources of semantic errors. The two introduced extensions represent the standard ways to prevent unintentional loss of allocated data.

The conclusion is therefore: as soon as a class is defined in which pointer data-members are used, a destructor, an overloaded assignment function and a copy constructor should be implemented.

# Chapter 7

# More About Operator Overloading

Now that we've covered the overloaded assignment operator in depth, and now that we've seen some examples of other overloaded operators as well (i.e., the insertion and extraction operators), let's take a look at some other interesting examples of operator overloading.

## 7.1 Overloading operator[]()

As our next example of operator overloading, we present a class which is meant to operate on an array of ints. Indexing the array elements occurs with the standard array operator [], but additionally the class checks for boundary overflow. Furthermore, the array operator is interesting in that it both *produces* a value and *accepts* a value, when used, respectively, as a *right-hand value* and a *left-hand value* in expressions.

An example of the use of the class is given here:

```
int main()
{
    IntArray
        x(20);              // 20 ints

    for (int i = 0; i < 20; i++)
        x[i] = i * 2;       // assign the elements

                            // produces boundary
                            // overflow
    for (int i = 0; i <= 20; i++)
        cout << "At index " << i << ": value is " << x[i] << endl;

    return (0);
}
```

This example shows how an array is created to contain 20 ints. The elements of the array can be assigned or retrieved. The above example should produce a run-time error, generated by the class

`IntArray`: the last `for` loop causing a boundary overflow, since `x[20]` is addressed while legal indices range from 0 to 19, inclusive.

We give the following class interface:

```
class IntArray
{
     public:
        IntArray(int size = 1);     // default size: 1 int
        IntArray(IntArray const &other);
        ~IntArray();
        IntArray const &operator=(IntArray const &other);

                        // overloaded index operators:
        int &operator[](int index);         // first
        int operator[](int index) const;    // second
    private:
        void boundary(int index) const;
        void destroy();             // standard functions
                                    // used to copy/destroy
        void copy(IntArray const &other);

        int
            *data,
            size;
};

#include <iostream>
```

Concerning this class interface we remark:

- The class has a constructor with a default `int` argument, specifying the array size. This function serves also as the default constructor, since the compiler will substitute 1 for the argument when none is given.
- The class internally uses a pointer to reach allocated memory. Hence, the necessary tools are provided: a copy constructor, an overloaded assignment function 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 obtain the elements of the `IntArray` object.

  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]` on the left-hand side *and* on the right-hand side of an assignment.

  We can therefore use the same function to retrieve and to assign values. Furthermore note that the returnvalue of the overloaded array operator is *not* an `int const &`, but rather an `int &`. In this situation we don't want the `const`, as we must be able to change the element we want to access, if the operator is used as a left-hand value in an assignment.

  However, this whole scheme fails if there's nothing to assign. Consider the situation where we have an `IntArray const stable(5);`. Such an object is a *const* object, which cannot be modified. The compiler detects this and will refuse to compile this object definition if only the first overloaded index operator is available. Hence the second overloaded index operator. Here the return-value is an `int`, rather than an `int &`, and the member-function itself is a `const` member function. This second form of the overloaded index operator cannot be used with *non*-`const` objects, but it's perfect for `const` objects. It can only be used for value-retrieval, not for value-assignment, but that is precisely what we want with `const` objects.

- We used the standard implementations of the copy constructor, the overloaded assignment operator and the destructor, discussed before (in section 6.4.1), albeit that we've left out the implementation of the function `destroy()`, as this function would consist of merely one statement (`delete data`).
- As the elements of `data` are `int`s, no `delete []` is needed. It does no harm, either. Therefore, since we use the `[]` when the object is created, we also use the `[]` when the data are eventually destroyed.

The member functions of the class are presented next.

```
#include "intarray.h"

IntArray::IntArray(int sz)
{
    if (sz < 1)
    {
        cerr << "IntArray: size of array must be >= 1, not " << sz
             << "!" << endl;
        exit(1);
    }
    // remember size, create array
    size = sz;
    data = new int [sz];
}

// copy constructor
IntArray::IntArray(IntArray const &other)
{
    copy(other);
}

// destructor
IntArray::~IntArray()
{
    delete [] data;
}

// overloaded assignment
IntArray const &IntArray::operator=(IntArray const &other)
{
    // take action only when no auto-assignment
    if (this != &other)
    {
        delete [] data;
        copy(other);
```

```
    }
    return (*this);
}

// copy() primitive
void IntArray::copy(IntArray const &other)
{
    // set size
    size = other.size;

    // create array
    data = new int [size];

    // copy other's values
    for (register int i = 0; i < size; i++)
        data[i] = other.data[i];
}

// here is the first overloaded array operator
int &IntArray::operator[](int index)
{
    boundary(index);
    return (data[index]);   // emit the reference
}

// and the second overloaded array operator
int IntArray::operator[](int index) const
{
    boundary(index);
    return (data[index]);   // emit the value
}

// the function checking the boundaries for the index:
void IntArray::boundary(int index) const
{
    // check for array boundary over/underflow
    if (index < 0 || index >= size)
    {
        cerr << "IntArray: boundary overflow or underflow, index = "
             << index << ", should range from 0 to " << size - 1 << endl;
        exit(1);
    }
}
```

## 7.2 Overloading operator new(size_t)

If the `operator new` is overloaded, it must have a `void *` return type, and at least an argument of type `size_t`. The `size_t` type is defined in `stddef.h`, 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();
```

Furthermore, the `new []` construction will always use the default operator `new`.

An example of the overloaded operator `new` for the class `X` is the following:

```
#include <stddef.h>

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 the operator `new` is defined for the `class X`. Assume that class is defined as follows[1]:

```
class X
{
    public:
        void *operator new(size_t sizeofX);

        int
            x,
            y,
            z;
};
```

Now, consider the following program fragment:

```
#include "X.h"  // class X interface etc.

int main()
{
    X
        *x = new X();

    cout << x->x << ", " << x->y << ", "<< x->z << endl;
```

[1]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`.

```
    return (0);
}
```

This small program produces the following output:

```
0, 0, 0
```

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 the `new` operator the allocated `X` object was already initialized to zeros when the constructor was called.

Non-static object member functions are passed a (hidden) pointer to the object on which they should operate. This hidden pointer becomes the `this` pointer inside the member function. This procedure is also followed by the constructor. In the following fragments of pseudo **C++** the pointer is made visible. In the first part an `X` object is declared 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
            // the compiler made 'x' available

void        // ask new to allocate the memory for an X
    *ptr = X::operator new();
X::X(ptr);  // and let the constructor operate on the
            // memory returned by 'operator new'
```

Notice that in the pseudo `C++` fragment the member functions were treated as static functions of the class `X`. Actually, the `operator new()` operator *is* a static functions 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 first. It can do that by allocating enough memory, and by initializing the area as required. Next, the memory is passed over to the constructor (as the `this` pointer) for further processing. The fact that an overloaded operator `new` is in fact 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);

        return (0);
    }
```

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 again is the `size_t` parameter, other parameters must be passed during the call to the `operator new`. For example:

```
    class X
    {
        public:
            void *operator new(size_t p1, unsigned p2);
            void *operator new(size_t p1, char const *fmt, ...);
    };

    int main()
    {
        X
            *object1 = new(12) X(),
            *object2 = new("%d %d", 12, 13) X(),
            *object3 = new("%d", 12) X();

        return (0);
    }
```

The object (object1) 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 object (object2) 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 `object3` also uses the second overloaded `operator new()`: that overloaded operator accepts a variable number of arguments, the first of which is a `char const *`.

## 7.3 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 returntype 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);
```

The 'home-made' `delete` operator is called after executing the class' destructor. 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 itself.
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)
{
    // ... whatever else is considered necessary

    // use the default operator delete
    ::delete ptr;
}
```

## 7.4 Cin, cout, cerr and their 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 $<<$. Adaptating a class in such a way that the `istream`'s extraction operator $>>$ can be used occurs in a similar way and is not further illustrated here.

The implementation of an overloaded operator $<<$ in the context of `cout` or `cerr` involves the base class of `cout` or `cerr`, which is `ostream`. This class is declared in the header file `iostream` and defines only overloaded operator functions for 'basic' types, such as, `int`, `char*`, etc.. The purpose of this section is to show how an operator function can be defined which processes a new class, say `Person` (see chapter 6.1) , so that constructions as the following one become possible:

```
Person
    kr("Kernighan and Ritchie", "unknown", "unknown");

cout << "Name, address and phone number of Person kr:\n"
     << kr
     << '\n';
```

The statement `cout << kr` involves the operator << and its two operands: an `ostream &` and a `Person &`. The proposed action is defined in a class-less operator function `operator<<()` expecting two arguments:

```
// declaration in, say, person.h
ostream &operator<<(ostream &, Person const &);

// definition in some source file
ostream &operator<<(ostream &stream, Person const &pers)
{
    return
    (
        stream << "Name:    " << pers.getname()
               << "Address: " << pers.getaddress()
               << "Phone:   " << pers.getphone()
    );
}
```

Concerning this function we remark the following:

- The function must return a (reference to) `ostream` object, to enable 'chaining' of the operator.
- The two operands of the operator << are stated as the two arguments of the overloading function.
- The class `ostream` provides the member function `opfx()`, which flushes any other `ostream` streams tied with the current stream. `opfx()` returns 0 when an error has been encountered (Cf. chapter 12).

  An improved form of the above function would therefore be:

  ```
  ostream &operator<<(ostream &stream, Person const &pers)
  {
      if (! stream.opfx())
          return (stream);
      ...
  }
  ```

## 7.5 Conversion operators

A class may be constructed around a basic type. E.g., it is often fruitful to define a class `String` around the `char *`. Such a class may define all kinds of operations, like assignments. Take a look at the following class interface:

```
class String
{
    public:
        String();
        String(char const *arg);
        ~String();
        String(String const &other);
        String const &operator=(String const &rvalue);
        String const &operator=(char const *rvalue);
    private:
        char
            *string;
};
```

Objects from this class can be initialized from a `char const *`, and also from a `String` itself. There is an overloaded assignment operator, allowing the assignment from a `String` object and from a `char const *`[2].

Usually, in classes that are less directly linked to their data than this `String` class, there will be an accessor member function, like `char const *String::getstr() const`. However, in the current context that looks a bit awkward, but it *also* doesn't seem to be the right way to go when an array of strings is defined, e.g., in a class `StringArray`, in which the `operator[]` is implemented to allow the access of individual strings. Take a look at the following class interface:

```
class StringArray
{
    public:
        StringArray(unsigned size);
        StringArray(StringArray const &other);
        StringArray const &operator=(StringArray const &rvalue);
        ~StringArray();

        String &operator[](unsigned index);
    private:
        String
            *store;
        unsigned
            n;
};
```

[2]Note that the assingment from a `char const *` also includes the null-pointer. An assignment like `stringObject = 0` is perfectly in order.

The `StringArray` class has one interesting member function: the overloaded array operator `operator[]`. It returns a `String` reference.

Using this operator assignments between the `String` elements can be realized:

```
StringArray
    sa(10);

... // assume the array is filled here

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";
```

When this is evaluated, the following steps are followed:

- 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;

cp = sa[3];
```

Well, this won't work: we would need an overloaded assignment operator for the 'class char const *'. However, there isn't such a class, and therefore we can't build that overloaded assignment operator (see also section 7.9). Furthermore, *casting* won't work: the compiler doesn't know how to cast a `String` to a `char const *`. How to proceed?

The naive solution is to resort to the accessor member function `getstr()`:

```
cp = sa[3].getstr();
```

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 *`. The general form of a conversion operator in the class interface is:

```
operator <type>();
```

With our `String` class, it would therefore be:

```
operator char const *();
```

The implementation of the conversion operator is straightforward:

```
String::operator char const *()
{
    return (string);
}
```

Notes:

- There is *no* mentioning of a return type. The conversion operator has the type of the returned value just after the `operator` keyword.
- In certain situations the compiler needs a hand to disambiguate our intentions. In a statement like

  ```
  printf("%s", sa[3]);
  ```

  the compiler is confused: are we going to pass a `String &` or a `char const *` to the `printf()` function? To help the compiler out, we supply an explicit cast here:

  ```
  printf("%s", static_cast<char const *>(sa[3]));
  ```

For completion, the final `String` class interface, containing the conversion operator, looks like this:

```
class String
{
    public:
        String();
        String(char const *arg);
        ~String();
        String(String const &other);
        String const &operator=(String const &rvalue);
        String const &operator=(char const *rvalue);
        operator char const *();
    private:
        char
            *string;
};
```

## 7.6 The 'explicit' keyword

Assume we have a class that's doing all kinds of interesting stuff. Its public members could be, e.g.:

```
class Convertor
{
    public:
        Convertor();
        Convertor(char const *str);
        Convertor(Convertor const &other);
        ~Convertor();
        operator char const*();
        void anyOtherMemberFunction();
};
```

Objects of the class `Convertor` may be constructed using a default constructor and using a `char const *`. Functions might return `Convertor` objects and functions might expect `Convertor` objects as arguments. E.g.,

```
Convertor returnConvertorObject()
{
    Convertor
        convertor;

    return (convertor);
}
void expectConvertorObject(Convertor const &object)
{
    ...
}
```

In cases like these, *implicit conversions* to `Convertor` objects will be performed if there are constructors having one parameter (or multiple parameters, using default argument values), if an argument of the type of the single parameter is passed to or returned from the function. E.g., the following function expects a `char const *` and returns an `Convertor object` due to the implicit conversion from `char const *` to `Convertor` using the `Convertor(char const *)` constructor as middleman:

```
Convertor returnConvertorObject(char const *str)
{
    return (str);
}
```

This conversion generally occurs wherever possible, and acts like some sort of 'reversed' conversion operator: in applicable situations the constructor expecting one argument will be used if the argument is specified, and the class object is required.

If such implicit use of a constructor is not appropriate, it can be prevented by using the `explicit` modifier with the constructor. Constructors using the `explicit` modifier can only be used for the explicit definition of objects, and cannot be used as implicit type convertors anymore. For example, to prevent the implicit conversion from `char const *` to `Convertor` the class interface of the class `Convertor` must contain the constructor

```
explicit Convertor(char const *str);
```

## 7.7 Overloading the increment and decrement operators

Overloading the increment (and decrement) operator creates a small 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`).

Suppose we define a class `bvector` whose members can be used to visit the elements of an array. The `bvector` object will return a pointer to an element of the array, and the increment operators will change the pointer to the next element. A partially defined `bvector` class is:

```
class bvector
{
    public:
        bvector(int *vector, unsigned size)
        :
            vector(vector),
            current(vector),
            finish(vector + size)
        {}
        int *begin()
        {
            return(current = vector);
        }
        operator int *() const
        {
            return (current);
        }
        // increment and decrement operators: see the text
    private:
        int
            *vector,
            *current,
            *finish;
};
```

In order to privide this class with an overloaded increment operator, the following overloaded `operator++()` can be designed:

```
int *bvector::operator++()
{
    return (++current);
}
```

As `current` is incremented before it is returned, the above overloaded `operator++()` clearly behaves like the prefix operator. However, it is not possible to use the same function to implement the postfix operator, as overloaded functions must differ in their parameterlists. To solve this problem, the convention is adopted to provide the *postfix* operator with an anonymous `int` parameter. So, the postfix increment operator can be designed as follows:

```
int *bvector::operator++(int)
{
    return (current++);
}
```

In situations where the function `operator++()` is called explicitly, a dummy `int` argument may be passed to the function to indicate that the postfix version is required. If no argument is provided, the prefix version of the operator is used. E.g.,

```
bvector
    *bvp = new bvector(intArray, 10);

bvp->operator++(1); // postfix operator++()
bvp->operator++()   // prefix operator++()
```

## 7.8 Function Objects

*Function Objects* are created by overloading the *function call* operator `operator()`. By defining the function call operator an object may be used as a function, hence the term *function objects*.

Function objects play an important role in the *generic algorithms* and they can be used profitably as alternatives to using pointers to functions. The fact that they are important in the context of the generic algorithms constitutes some sort of a didactical dilemma: at this point it would have been nice if the generic algorithms would have been covered, but for the discussion of the generic algorithms knowledge of function objects is an advantage. This bootstrap problem is solved in a well known way: by ignoring the dependency.

Function objects are `class type` objects for which the `operator()` has been defined. Usually they are used in combination with the generic algorithms, but they are also used in situations where otherwise pointers to functions would have been used. Another reason for using function objects is to support `inline` functions, something that is not possible via the pointers to functions construction.

Assume we have a class `Person` and an array of `Person` objects. 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 in which this function is used is, e.g.,

```
Person
    *pArray;
unsigned
    n;

n = fillPerson(&pArray);

Person
    target(...);

cout <<
    "The target person is " <<
    (
        lsearch(&target, pArray, &n, sizeof(Person), compareFunction) ?
            "found"
        :
            "not found"
    ) <<
    endl;
```

The function `fillPerson()` is called to fill the array, the target person is defined, and then `lsearch()` is used to locate the target person. The comparison function must be available, as its address is passed over to the 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 operator is available as well. In this situation an *inline* compare function cannot be used: as the address of the `compare()` function must be known to the `lsearch()` function. So, on the average `n / 2` times *at least* the following actions take place:

- The two arguments of the comparefunction are pushed on the stack,
- The final parameter of `lsearch()` is evaluated, producing the address of `compareFunction()`,
- The comparefunction is called,
- The address of the right-hand argument of the `Person::operator!=())` argument is pushed on the stack,

- The `operator!=()` function is evaluated,
- The argument of `Person::operator!=()`) argument is popped off the stack,
- The two arguments of the comparefunction are popped off the stack.

When using function objects a different picture emerges. Assume we have constructed a function `PersonSearch()`, having the following prototype (realize that this is not the real thing. Normally a generic algorithm will be used instead of a home-made function. But for now our `PersonSearch()` function is used for the sake of argument):

```
Person const *PersonSearch(Person *base, size_t nmemb,
                           Person const &target);
```

The next program fragment shows the use of this function:

```
Person
    *pArray;
unsigned
    n;

n = fillPerson(&pArray);

cout <<
    "The target person is " <<
    (
        PersonSearch(pArray, n, Person(...)) ?
            "found"
        :
            "not found"
    ) <<
    endl;
```

Here we see that the target person is passed over to the function using an anonymous `Person` object. A named object could have been used as well, though. What happens inside `PersonSearch()` is shown next:

```
Person const *PersonSearch(Person *base, size_t nmemb,
                           Person const &target)
{
    for (int idx = 0; idx < nmemb; ++idx)
        if (!target(base[idx]))     // using the same returnvalues
            return (base + idx);    // as lsearch(): 0 means 'found'
    return (0);
}
```

The expression `target(base[idx])` shows our `target` object being used as a function object. Its implementation can be something like:

```
int Person::operator()(Person const &other) const
{
    return (*this != other);
}
```

Note the somewhat peculiar syntax: `operator()(...)`. The first set of parentheses define the particular operator that is overloaded: the function call operator. The second set of parentheses define the parameters that are required for this function. The `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, and we could consider making it inline. Assuming we do so, here is what happens when the `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.

The `operator()` could have been avoided altogether in the above example. However, in the coming sections several predefined function objects are introduced calling specific operators of underlying datatypes. Usually these function object will receive one or two arguments (for, respectively, unary and binary operators).

Function objects play important roles in combination with generic algorithms. For example, there exists a generic algorithm `sort` that takes two iterators defining the range of objects that should be sorted, and a function object calling the appropriate comparison operator for two objects. Let's take a quick look at this situation. Assume strings are stored in a vector, and we want to sort the vector in descending order. In that case, sorting the vector `stringVec` is as simple as:

```
sort(stringVec.begin(), stringVec.end(), greater<string>());
```

The last argument is in fact a *constructor* of the `greater` (template) class applied on `strings`. This object is called (as function object) by the `sort()` generic algorithm. The function object itself is *not* visible at this point: don't confuse the parentheses in `greater<string>()` with the calling of the function object. When the function object is actually called, it receives two arguments: two strings to compare for 'greaterness'. Internally, the `operator>()` of the underlying datatype (i.e., `string`) is called to compare the two objects. Since the `greater::operator()` is defined *inline*, it is not actually present in the code. Rather, the `string::operator>()` is called by `sort()`.

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 in increasing order its command-line arguments:

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>
#include <string.h>

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;

    return (0);
}
```

The default constructor of the `class case_less` is used with the final argument of `sort()`. 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 provide it with two `string` arguments, which are used in the `strcasecmp()` function. Furthermore, the `operator()` function is made inline, so that it does not produce overhead in the `sort()` function. The `sort()` function calls the function object with various combinations of `strings`, i.e., it *thinks* it does so. However, in fact it calls `strcasecmp()`, due to the inline-nature of `case_less::operator()`.

The comparison function object is often a *predefined function object*, since these are available for most of the common operations.

A function object may be defined inline. This is not possible for functions that are called indirectly (i.e., via pointers to functions). So, even if the function object needs to do very little work it has to be defined as an ordinary function if it is going to be called via pointers. The overhead of performing the indirect call may not outweight 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 object may access the data of the objects for which they are called directly, as they have access to the private data of their object. In situations where a function must be able to serve many different datatypes (like the `qsort()` function) it is always somewhat cumbersome to reach the data of the involved objects via a pointer to a function of global scope.

In the following sections the available predefined function objects are presented, together with some examples showing their use. At the end of this section about function objects *function adaptors* are presented.

### 7.8.1 Categories of Function objects

Function objects may be defined when necessary. However, it is also (and often) possible to use predefined function objects. In order to use the predefined function objects the header file `functional` must be included:

```
        #include <functional>
```

The predefined function objects are used predominantly with the generic algorithms. Predefined function objects exists for arithmetic, relational, and logical functions. They are discussed in the coming sections.

#### Arithmetic Function Objects

The arithmetic function objects support the standard arithmetic operations: addition, subtraction, multiplication, division, modulus and negation. By using the predefined function objects, the corresponding operator of the associated data type is invoked. For example, for addition the function object `plus<Type>` is available. If we set `type` to `unsigned` then the `+` operator for unsigneds is used, if we set `type` to `string`, then the `+` operator for strings is used. For example:

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

int main(int argc, char **argv)
{
    plus<unsigned>
        uAdd;       // function object to add unsigneds

    cout << "3 + 5 = " << uAdd(3, 5) << endl;

    plus<string>
        sAdd;       // function object to add strings

    cout << "argv[0] + argv[1] = " << sAdd(argv[0], argv[1]) << endl;
}
```

Why is this useful? Note that the function object can be used for all kinds of data types, not only on the predefined datatypes, but on any (class) type 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 each element of an array in turn. E.g., we want to compute the sum of the elements of an array, or we want to concatenate all the strings in a text-array. In situations like these the function objects come in handy. As noted before, the function objects are most heavily used in the context of the generic algorithms, so let's take a quick look 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 its command line arguments, and prints the final string:

```
#include <iostream>
#include <string>
#include <functional>
#include <numeric>

int main(int argc, char **argv)
{
    string
        result =
            accumulate(argv, argv + argc, string(""), plus<string>());

    cout << "All concatenated arguments: " << result << endl;
}
```

The first two arguments define the (iterator) range of elements to visit, the third argument is `string("")`. This anonymous string object provides an initial value. It could as well have been initialized to

```
string("All concatenated elements:  ")
```

in which case the `cout` statement could have been a simple

```
cout << result << endl
```

Then, the operator to apply is `plus<string>()`. Here it is important to note the function call notation: it is *not* `plus<string>`, but rather `plus<string>()`. The final concatenated string is returned.

Now we define our own class data type `Time`, in which the `operator+()` has been overloaded. Again, we can apply the predefined function object `plus`, now tailored to our newly defined datatype, to add times:

```
#include <iostream>
#include <strstream>
#include <string>
#include <vector>
#include <functional>
```

```
#include <numeric>

class Time
{
    public:
        Time(unsigned hours, unsigned minutes, unsigned seconds)
        {
            days = 0;
            this->hours   = hours;
            this->minutes = minutes;
            this->seconds = seconds;
        }
        Time(Time const &other)
        {
            this->days    = other.days;
            this->hours   = other.hours;
            this->minutes = other.minutes;
            this->seconds = other.seconds;
        }
        Time const operator+(Time const &rValue) const
        {
            Time
                added(*this);
            added.seconds   += rValue.seconds;
            added.minutes   += rValue.minutes   + added.seconds / 60;
            added.hours     += rValue.hours     + added.minutes / 60;
            added.days      += rValue.days      + added.hours   / 24;
            added.seconds   %= 60;
            added.minutes   %= 60;
            added.hours     %= 24;
            return (added);
        }
        operator char const *() const
        {
            static ostrstream
                timeString;

            timeString.seekp(ios::beg);
            timeString << days << " days, " << hours << ":" <<
                        minutes << ":" << seconds << ends;
            return (timeString.str());
        }
    private:
        unsigned
            days,
            hours,
            minutes,
            seconds;
};

int main(int argc, char **argv)
{
    vector<Time>
```

```
        tvector;

    tvector.push_back(Time( 1, 10, 20));
    tvector.push_back(Time(10, 30, 40));
    tvector.push_back(Time(20, 50,  0));
    tvector.push_back(Time(30, 20, 30));

    cout <<
        accumulate
        (
            tvector.begin(), tvector.end(),
            Time(0, 0, 0), plus<Time>()
        ) << endl;

}
```

Note that all member functions of `Time` in the above source are inline functions. This approach was followed in order to keep the example relatively small, and to show explicitly that the `operator+()` function may be an inline function. On the other hand, in real life the `operator+()` function of `Time` should probably not be made inline, due to its size. Considering the previous discussion of the `plus` function object, the example is pretty straightforward. The class `Time` defines two constructors, the second one being the copy-constructor, it defines a conversion operator (operator char const *()) to produce a textual representation of the stored time (deploying an `ostrstream` object, see chapter 12), and it defines its own `operator+()`, adding two time objects.

The organization of the `operator+()` deserves some attention. In expressions like `x + y` neither `x` nor `y` are modified. The result of the addition is returned as a temporary value, which is then used in the rest of the expression. Consequently, in the `operator+()` function the `this` object and the `rValue` object must not be modified. Hence the `const` modifier for the function, forcing `this` to be constant, and the `const` modifier for `rValue`, forcing `rValue` to be constant. The sum of both times is stored in a separate `Time` object, a copy of which is then returned by the function.

In the `main()` function four `Time` objects are stored in a `vector<Time>` object. Then, the `accumulate()` generic algorithm is called to compute the accumulated time. It returns a `Time` object, which cannot be inserted in the `cout ostream` object. Fortunately, the conversion operator is available, and this conversion operator is called implicitly to produce the required `char const *` string from the `Time` object returned by the `accumulate()` generic algorithm.

While the first example did show the use of a *named* function object, the last two examples showed *unnamed* or *anonymous* objects which were passed to the (`accumulate`) function.

The following arithmetic objects are available as predefined objects:

- `plus`, as shown this object calls the `operator+()`
- `minus`, calling `operator-()` as a binary operator,
- `multiplies`, calling `operator*()` as a binary operator,

- `divides`, calling `operator/()`,
- `modulus`, calling `operator%()`,
- `negate`, calling `operator-()` as a unary operator.

An example using the unary `operator-()` is the following, in which the `transform()` generic algorithm is used to toggle the signs of all elements in an array. The `transform()` generic algorithm expects two iterators, defining the range of objects to be transformed, an iterator defining the begin of the destination range (which may be the same iterator as the first argument) and a function object defining a unary operation for the indicated data type.

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>

int main(int argc, char **argv)
{
    int
        iArr[] = { 1, -2, 3, -4, 5, -6 };

    transform(iArr, iArr + 6, iArr, negate<int>());

    for (int idx = 0; idx < 6; ++idx)
        cout << iArr[idx] << ", ";

    cout << endl;
}
```

### Relational Function Objects

The relational operators may be called from the relational function objects. All standard relational operators are supported: `==`, `!=`, >, `>=`, < and <=. The following objects are available:

- equal_to<Type>, calling `operator==()`,
- not_equal_to<Type>, calling `operator!=()`,
- greater<Type>, calling `operator>()`,
- greater_equal<Type>, calling `operator>=()`,
- less<Type>, calling `operator<()`,
- less_equal<Type>, calling `operator<=()`.

Like the arithmetic function objects, these function objects can be used as *named* and *unnamed* objects. An example using the relational function objects using the generic algorithm `sort()` is:

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>

int main(int argc, char **argv)
{
    sort(argv, argv + argc, greater_equal<string>());
    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << endl;

    sort(argv, argv + argc, less<string>());
    for (int idx = 0; idx < argc; ++idx)
        cout << argv[idx] << " ";
    cout << endl;

    return (0);
}
```

The `sort()` generic algorithm expects an iterator range and a comparator object for the underlying data type. The example shows the alphabetic sorting of strings and the reversed sorting of strings. By passing `greater_equal<string>()` the strings are sorted in *decreasing* order (the first word will be the 'greatest'), by passing `less<string>()` the strings are sorted in *increasing* order (the first word will be the 'smallest').

Note that the type of the elements of `argv` is `char *`, and that the relational function object expects a `string`. The relational object `greater_equal<string>()` will therefore use the >= operator of strings, but will be called with `char *` variables. The conversion from `char *` arguments to `string const &` parameters is done implicitly by the `string(char const *)` constructor.

### Logical Function Objects

The logical operators are called by the logical function objects. The standard logical operators are supported: `&&`, `||` and `!`. The following objects are available:

- `logical_and<Type>`, calling `operator&&()`,
- `logical_or<Type>`, calling `operator||()`,
- `logical_not<Type>`, calling `operator!()` (unary operator).

An example using the `operator!()` is the following trivial example, in which the `transform()` generic algorithm is used to transform the logical values stored in an array:

```
#include <iostream>
#include <string>
#include <functional>
#include <algorithm>
```

```
int main(int argc, char **argv)
{
    bool
        bArr[] = {true, true, true, false, false, false};
    unsigned const
        bArrSize = sizeof(bArr) / sizeof(bool);

    for (int idx = 0; idx < bArrSize; ++idx)
        cout << bArr[idx] << " ";
    cout << endl;

    transform(bArr, bArr + bArrSize, bArr, logical_not<bool>());

    for (int idx = 0; idx < bArrSize; ++idx)
        cout << bArr[idx] << " ";
    cout << endl;

    return (0);
}
```

### 7.8.2 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 fixed function object. For example, with the `minus<int>` function object, which is a binary function object, the first argument may be fixed to 100, meaning that the resulting value will always be `100` minus the value of the second argument. Either the first or the second argument may be bound to a specific value. To bind the first argument to a specific value, the function object `bind1st()` is used. To bind the second argument of a binary function to a specific value `bind2nd()` is used. As an example, assume we want to count all elements of a vector of `Person` objects that exceed (according to some criterion) some reference `Person` object. For this situation we pass the following binder and relational function object to the `count_if()` generic algorithm:

  ```
  bind2nd(greater<Person>(), referencePerson)
  ```

  The `count_if()` generic algorithm visits all the elements in an iterator-range, returning the number of times the predicate specified in its final argument returns `true`. Each of the elements of the iterator range is given to the predicate, which is therefore a unary function. By using the binder the binary function object `greater()` is adapted to a unary function object, comparing each of the elements in the range to the reference person. Here is, to be complete, the call of the `count_if()` function:

  ```
  count_if(pVector.begin(), pVector.end(),
      bind2nd(greater<Person>(), referencePerson))
  ```

- The *negators* are function adaptors converting the truth value of a predicate function. Since there are unary and binary predicate functions, there are two negator function adaptors: `not1()` is the negator to be used with unary function adaptors, `not2()` is the negator to be used with binary function objects.

If we want to count the number of persons in a `vector<Person>` vector *not* exceeding a certain reference person, we may, among other approaches, use either of the following alternatives:

- Use a binary predicate that directly offers the required comparison:

```
        count_if(pVector.begin(), pVector.end(),
            bind2nd(less_equal<Person>(), referencePerson))
```

- Use `not2` in combination with the `greater()` predicate:

```
        count_if(pVector.begin(), pVector.end(),
            bind2nd(not2(greater<Person>()), referencePerson))
```

- Use `not1` in combination with the `bind2nd()` predicate:

```
        count_if(pVector.begin(), pVector.end(),
            not1(bind2nd((greater<Person>()), referencePerson)))
```

The following small example illustrates the use of the negator function adaptors, completing the section on function objects:

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>

int main(int argc, char **argv)
{
    int
       iArr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    cout << count_if(iArr, iArr + 10, bind2nd(less_equal<int>(), 6)) <<
        endl;
    cout << count_if(iArr, iArr + 10, bind2nd(not2(greater<int>()), 6)) <<
        endl;
    cout << count_if(iArr, iArr + 10, not1(bind2nd(greater<int>(), 6))) <<
        endl;

    return (0);
}
```

## 7.9 Overloadable Operators

The following operators can be overloaded:

```
+      -      *      /      %      ^      &      |
~      !      ,      =      <      >      <=     >=
++     --     <<     >>     ==     !=     &&     ||
+=     -=     *=     /=     %=     ^=     &=     |=
<<=    >>=    []     ()     ->     ->*    new    delete
```

However, some 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, as we have seen in section 7.5.

# Chapter 8

# 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*.

The abstract containers rely heavily on *templates*, which are covered near the end of the **C++** Annotations, in chapter 17. However, in order to use the abstract containers, only a minimal grasp of the template concept is needed. In **C++** a *template* is in fact a recipe for constructing a function or a complete class. The recipe tries to abstract the functionality of the class or function as much as possible from the data on which the class or function operate. As the types of the data on which the templates operate were not known by the time the template was constructed, the datatypes are either inferred from the context in which a template function is used, or they are mentioned explicitly by the time a template class is used (the term that's used here is *instantiated*). In situations where the types are explicitly mentioned, the *angular bracket notation* is used to indicate which data types are required. For example, below (in section 8.1) we'll encounter the `pair` container, which requires the explicit mentioning of two data types. E.g., to define a `pair` variable containing both an `int` and a `string`, the notation

```
pair<int, string>
    myPair;
```

is used. Here, `myPair` is defined as a `pair` variable, containing both an `int` and a `string`.

The angular bracket notation is used intensively in the following discussion of the abstract container. Actually, understanding this part of templates is the only real requirement for being able to use the abstract containers. Now that we've introduced this notation, we can postpone the more thorough discussion of templates to chapter 17, and get on with their use in the form of the abstract container classes.

Most of the abstract containers are *sequential* containers: they represent a series of data which can be stored and retrieved in some sequential way. Examples are the `vector`, implementing an extendable array, the `list`, implementing a datastructure in which insertions and deletions can be easily realized, a `queue`, 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* datastructure.

Apart from the sequential containers, several special containers are available. The `pair` is a basic container in which a pair of values (of types that are left open for further specification) can be stored, like two strings, two ints, a string and a double, etc.. Pairs are often used to return data elements that naturally come in pairs. For example, the `map` is an abstract container in which keys and corresponding values are stored. Elements of these maps are returned as `pairs`.

A variant of the `pair` is the `complex` container, which implements 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 exists also an abstract container for the implementation of a *hashtable*, but that container is not (yet) accepted by the `ISO/ANSI` standard. The final section of this chapter will cover the hashtable to some extent.

All containers support the = operator to assign two containers of the same type to each other. All containers also support the `==, !=,` $<$`,` $<=$`,` $>$ and $>=$ operators.

Note that if a user-defined type (usually a `class`-type) is to be stored in a container, the user-defined type must 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 even more tasks than is possible with the containers themselves, like counting, filling, merging, filtering etc.. An overview of the generic algorithms and their applications is given in chapter 11. Generic algorithms usually rely on the availability of *iterators*, which represent begin and endpoints for processing data stored inside the containers. The abstract containers normally have constructors and members using iterators themselves, and they have members returning iterators (comparable to the `string::begin()` and `string::end()` members). In the remainder of this chapter the use of iterators is not really covered. Refer to chapter 11 for the discussion of iterators.

The url http://www.sgi.com/Technology/STL is worth visiting by those readers who want more information about the abstract containers and the standard template library than can be provided in the **C++** annotations.

Containers often collect data during their lifetime. When a container goes out of scope, its destructor tries to destroy its data elements. This only succeeds if the data elements themselves are stored inside the container. If the data elements of containers are pointers, the data to which these pointers point will not be destroyed, and a memory leak will result. A consequence of this scheme is that the data stored in a container should be considered the 'property' of the container: the container should be able to destroy its data elements when the destructor of the container is called. Consequently, the container should not only contain no pointer data, but it should also not contain `const` data elements, as these data elements cannot be destroyed by the container's destructor.

## 8.1 The 'pair' container

The `pair` container is a rather basic container. It can be used to store two elements, called `first` and `second`, and that's about it. To define a variable as a `pair` container, the header file

```
#include <utility>
```

must be included.

The data types of a `pair` are defined when the `pair` variable is defined, using the standard template (see chapter `Templates`) notation:

```
pair<string, string>
    piper("PA28", "PH-ANI"),
    cessna("C172", "PH-ANG");
```

here, the variables `piper` and `cessna` are defined as `pair` variables containing two `strings`. Both strings can be retrieved using the `first` and `second` fields of the `pair` type:

```
cout << piper.first << endl <<  // shows 'PA28'
        cessna.second << endl;  // shows 'PH-ANG'
```

The `first` and `second` members can also be used to reassign values:

```
cessna.first = "C152";
cessna.second = "PH-ANW";
```

If a `pair` variable must be completely reassigned, it is also possible to use an *anonymous* pair variable as the right-hand side operand of the assignment. An anonymous variable defines a temporary variable (which receives no name) solely for the purpose of (re)assigning another variable of the same type. Its general form is

```
type(initializer list)
```

Note, however, that with a `pair` variable the type specification is not completed when the containername `pair` has been mentioned. It also requires the specification of the data types which are stored inside the pair. For this the (template) angular bracket notation is used again. E.g., the reassignment of the `cessna` pair variable could also have been accomplished as follows:

```
cessna = pair<string, string>("C152", "PH-ANW");
```

In cases like this, 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 amount of typing may be reduced and legibility might be improved by first defining a name for the clause, and then using the defined name later on. 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` has no further special features. It is, however, a basic ingredient of the upcoming abstract containers `map`, `multimap` and `hash_map`.

## 8.2 Sequential Containers

### 8.2.1 The 'vector' container

The `vector` class implements an (expandable) array. To use the `vector`, the header file `vector` must be included:

```
#include <vector>
```

`Vector`s can be used like arrays, and can be defined with a fixed number of elements. E.g., to define a vector of 30 `int`s we do

```
vector<int>
    iVector(30);
```

Note the specification of the data type that is to be used: the datatype is given between angular brackets after the `vector` container name. So, a vector of 30 `string`s is defined as

```
vector<string>
    strVector(30);
```

One of the nice characteristics of defining such a vector is that it's initialized to the data type's default value. If there is a default constructor, it is called to construct the elements of the vector. For the basic data types the initial value is zero. So, for the `int` vector we know its values are 0.

Another way to initialize the vector is to use explicit initialization values:

```
vector<int>
    iVector(1, 2, 3);
```

This does not work, however, if a vector of one element must be initialized to a non-default value.

As with `string` variables,

- `vector` objects may be initialized with other `vectors`, or parts of existing `vectors` may be used to initialize a `vector`:

```
vector<int>
    a(10);
...
vector<int>
    b(&a[3], &a[6]);
```

Note here that the last element mentioned is *not* used for the initialization. This is a simple example of the use of *iterators*, in which the range of values that is used starts at the first value, and includes all elements up to, but not including the last value mentioned. The standard notation for this is `[begin, end)`.

- `vectors` may be assigned to each other,
- the subscript operator may be used to retrieve individual elements,
- the `==` and `!=` operators may be used to test the equality of two `vectors`.
- the `<` operator may be used to test whether each element in the left-hand operand vector is less than each corresponding element in the right-hand operand vector. The `<=`, `>` and `>=` operators are also available.
- the `size()` and `empty()` member functions are available,
- the `swap()` member function is available, swapping two vectors. E.g.,

```
int main()
{
    vector<int>
        v1(10),
        v2(10);

    v1.swap(v2);
}
```

- elements may be inserted at a certain position `pos`. Below `source` represents a value of the type that is stored in the vector, while `pos` is an *iterator* pointing to a position in the vector where `source` must be inserted:

- `insert(pos, source)` inserts `source` at `pos`,
- `insert(pos, begin, end)` inserts the elements in the iterator range `[begin, end)`.
- `insert(pos, n, source)` inserts `n` elements having value `source` at position `pos`.

- elements may be erased:
  - `erase()` and `clear()` both erase all elements, `clear()` is not available with `strings`.
  - `erase(pos)` erases all elements starting at position `pos`,
  - `erase(begin, end)` erases elements indicated by the iterator range `[begin, end)`.
- `resize(n)` and `resize(n, source)` may be used to resize the vector to a size of `n`. If the vector is expanded, the extra elements are initialized by the default value of the used datatype, or by the explicitly provided value `source`.

Also available are:

- `void pop_back()` may be used to remove the last element from the vector. The element is *not* returned by this member function.
- `front()`, returning the initial element of the `vector`,
- `back()`, returning the final element of the `vector`,
- `push_back(source)` stores `source` at the end of the `vector`: a new element is added at the end.

Note that a vector may be defined without size: `vector<int> ivect;`. This defines an empty vector, without any element at all. Therefore, a statement like `ivect[0] = 18;` would (in this case) be an error, as there isn't any element as yet. In this case the preferred idiom is `ivect.push_back(18);`

### 8.2.2 The 'list' container

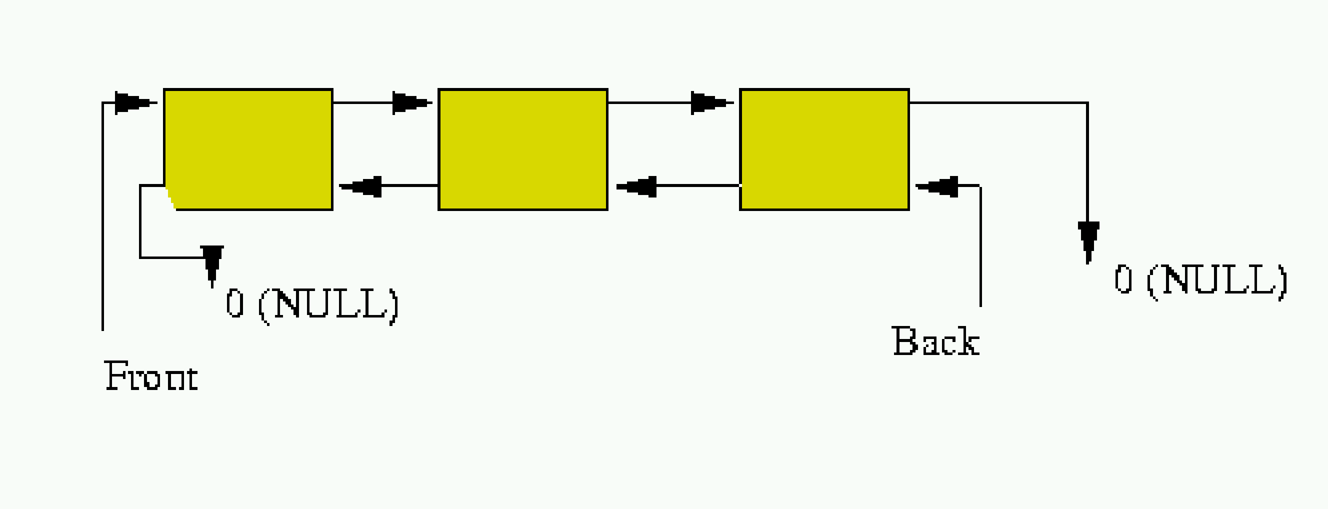
The `list` class implements a list datastructure. To use the `list`, the header file `list` must be included:

```
#include <list>
```

A list is depicted in figure 8.1.

In figure 8.1 it is shown that a list consists of separate data-items, connected to each other by pointers. The list can be traversed in two ways: starting at the *Front* the list may be traversed from left to right, until the 0-pointer is reached at the end of the rightmost data-item. The list can also be traversed from right to left: starting at the *Back*, the list is traversed from right to left, until eventually the 0-pointer emanating from the leftmost data-item is reached.

Both lists and vectors are often possible datastructures 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 datastructures must be made.




\

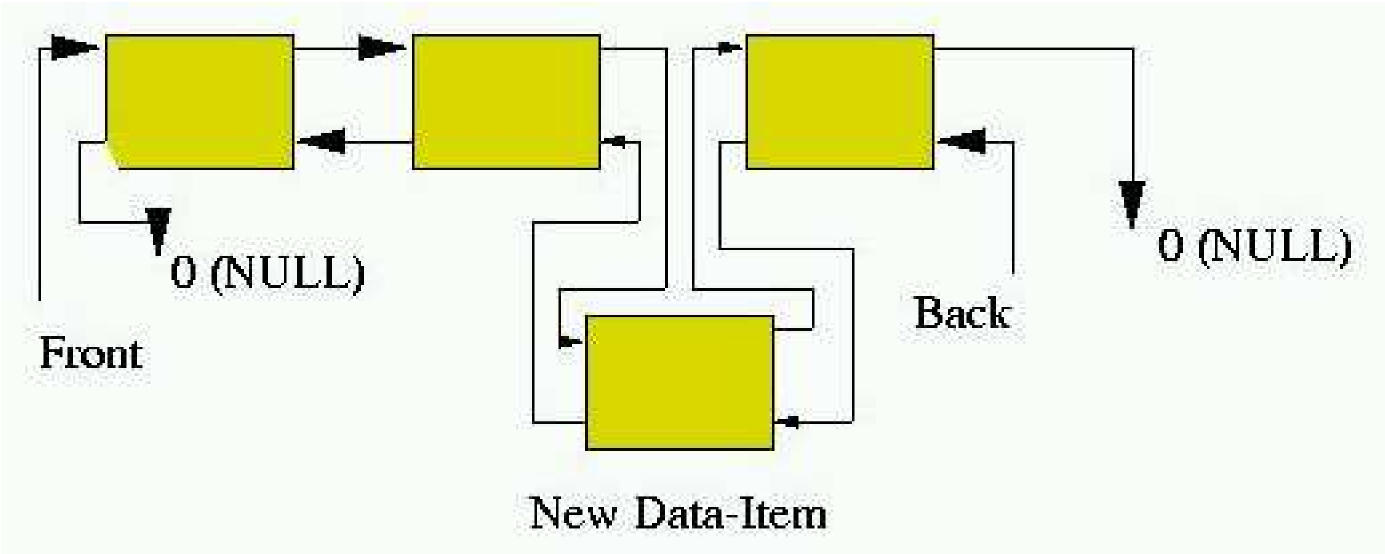
Figure 8.1: A list data-structure

- When the majority of accesses is random, then the `vector` is the preferred datastructure. E.g., in a program that counts the frequencies of characters in a textfile, a `vector<int> frequencies(256)` is the datastructure doing the trick, as the values of the received characters can be used as indices into the `frequencies` vector.
- The previous example illustrates a second rule of thumb, also favoring the `vector`: if the number of elements is known in advance (and does not notably change during the lifetime of the program), the vector is also preferred over the `list`.
- In cases where insertions and 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. Nonetheless, the `list` container exists, and it may become popular now that the list-management is part of the implementation of the abstract container.

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 dynamical growth does involve a lot of copying of data elements. Clearly, copying a million large datastructures 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 the remainder of the list structure: inserting a new element in a list merely requires us to juggle some pointers. In figure 8.2 this is shown: a new element is inserted between the second and third element, creating a new list of four elements.

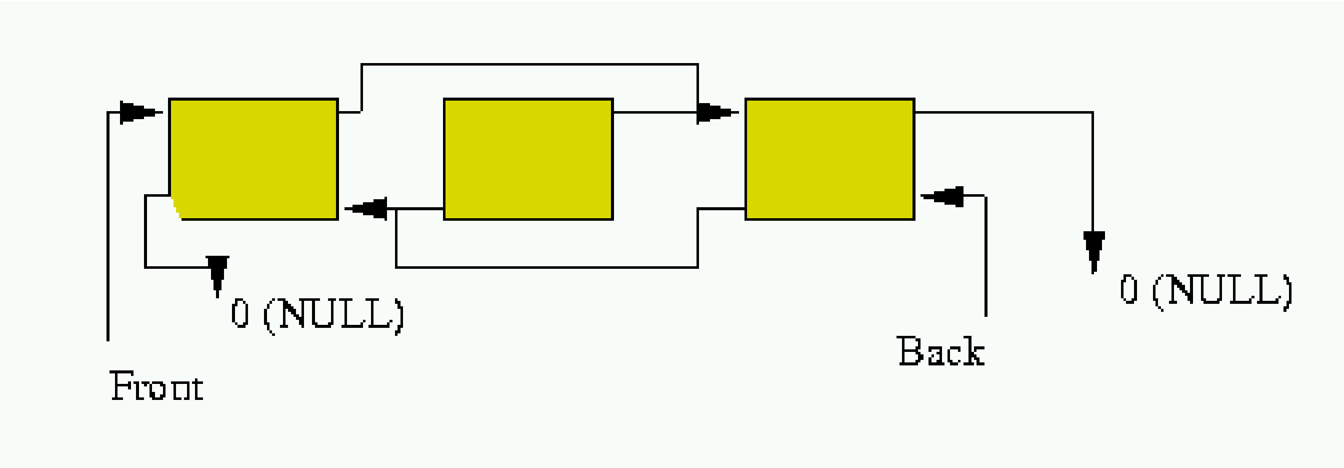
Removing an element from a list also is a simple matter. Starting again from the situation shown in figure 8.1, figure 8.3 shows what happens if element two is removed from our list. Again: only pointers need to be juggled. In this case it's even simpler than adding an element: only two pointers need 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 datastructure to prefer. There are rules of thumb, which may be adhered to. But if worse comes to worst, a profiler may be required to find out what's working best. But, no matter what thoughts remain, the `list` container is available, so




\

Figure 8.2: Adding a new element to a list




\

Figure 8.3: Removing an element from a list

let's see what we can do with it. As with the `vector`-class, the following constructors and member functions are available:
Constructors:

- an empty list is created using, e.g.,

```
list<string>
    strList;
```

- A list may be initialized with a certain number of elements. By default, if the initialization value is not explicitly mentioned, the defaultvalue or default constructor for the actual datatype is used. For example:

```
list<string>
    hello(5, string("Hello")),  // initialize to 5 Hello's
    zilch(10);                  // initialize to 10 empty strings
```

- A list may be initialized using a two iterators, e.g., 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>
    svector;
list<string>
    slist(&svector[5], &svector[11]);
```

Note that a list may be defined without size:

```
list<int> ivect;
```

This defines an empty list, without any element at all. So, a statement like

```
*ivect.begin() = 18;
```

would in this case be an error, as there isn't any element as yet. In this case, the preferred idiom is:

```
ivect.push_back(18);
```

Other member functions, some of which were also available in `vector`, are:

- `back()`, returning the last element of the list.
- `clear()`,
- `front()`, returning the first element of the list.

- `empty()`,
- elements may be erased:
  - `erase()` and `clear()` both erase all elements,
  - `erase(pos)` erases all elements starting at the position pointed to by iterator `pos`,
  - `erase(begin, end)` erases elements indicated by the iterator range `[begin, end)`.
- elements may be inserted at a certain position pointed to by the iterator `pos`:
  - `insert(pos, source)` inserts `source` at the position pointed to by `pos`,
  - `insert(pos, begin, end)` inserts the elements in the iterator range `[begin, end)` at the position pointed to by `pos`.
  - `insert(pos, n, argument)` inserts `n` elements having value `argument` at the position pointed to by `pos`. The data type of `argument` must be equal to the data type of the the elements of the list.
- `resize(n)` and `resize(n, argument)` may be used to resize the list to a size of `n`. If the list is expanded, the extra elements are initialized by the default value of the used datatype, or by the explicitly provided value `argument`.
- `size()`,
- `swap(argument)`, swaps two lists.

Also available are:

- `void push_front(source)` to enter a new element at the head of the list.
- `void push_back(source)` to enter a new element at the end of the list.
- `void pop_front()` may be used to remove the first element from the list. This element is *not* returned by this member function.
- `void pop_back()` may be used to remove the last element from the list. This element is *not* returned by this member function.
- `remove(source)`: This member function removes all occurrences of `source` from the list: the two strings `Hello` are removed from the list `object` in the following example:

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

int main()
{
    list<string>
        object;

    object.push_back(string("Hello"));
    object.push_back(string("World"));
    object.push_back(string("Hello"));
    object.push_back(string("World"));

    object.remove(string("Hello"));
```

```
    while (object.size())
    {
        cout << object.front() << endl;
        object.pop_front();
    }

    return (0);
}
```

- `sort()` will sort the list. Once the list has been sorted, the following member function (`unique()`) may be used to remove all multiply occurring elements from the list, leaving only one element of each. The following example shows the use of both member functions.
- `unique()` makes sure that each element will occur only once. Here's an example, leaving each single word only once in the list:

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

int main()
{
    list<string>
        target;

    target.push_back(string("A"));
    target.push_back(string("rose"));
    target.push_back(string("is"));
    target.push_back(string("a"));
    target.push_back(string("rose"));
    target.push_back(string("is"));
    target.push_back(string("a"));
    target.push_back(string("rose"));

    cout << "Initially we have: " << endl;
    list<string>::iterator
        from;
    for (from = target.begin(); from != target.end(); ++from)
        cout << *from << " ";
    cout << endl;
    target.sort();
    cout << "After sort() we have: " << endl;
    for (from = target.begin(); from != target.end(); ++from)
        cout << *from << " ";
    cout << endl;
    target.unique();
    cout << "After unique() we have: " << endl;
    for (from = target.begin(); from != target.end(); ++from)
        cout << *from << " ";
    cout << endl;
```

```
    return (0);
}
```

- `merge(argument)` combines the current list and the `argument` list. The merging will add elements of `source` to `target`. When both lists are ordered, the resulting list will be ordered as well. If both list are not completely ordered, the resulting list will be ordered as much as possible, given the initial ordering of the elements in each list. In the following example this is illustrated: the `object` list is not completely ordered, but the resulting list (`alpha bravo golf oscar mike november quebec zulu`) is ordered 'as much as possible': `mike` has to follow `oscar`, since this ordering is imposed by `object`, but given that imperfection the resulting list is ordered alphabetically.

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

int main()
{
    list<string>
        object,
        argument;

    object.push_back(string("alfa"));
    object.push_back(string("bravo"));
    object.push_back(string("golf"));
    object.push_back(string("quebec"));

    argument.push_back(string("oscar"));
    argument.push_back(string("mike"));
    argument.push_back(string("november"));
    argument.push_back(string("zulu"));

    object.merge(argument);

    list<string>::iterator
        from;
    for (from = object.begin(); from != object.end(); ++from)
        cout << *from << " ";
    cout << endl;

    return (0);
}
```

Note that the members `merge()` and `sort()` both assume the availability of the $<$ and `==` operators.

- `target.splice(iterator position, list source)`: This member function transfers the contents of `source` to the current list. Following `splice()`, `source` is empty. For example:

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

int main()
{
    list<string>
        object;

    object.push_front(string("Hello"));
    object.push_back(string("World"));

    list<string>
        argument(object);

    object.splice(++object.begin(), argument);

    cout << "Object contains " << object.size() << " elements, " <<
            "Argument contains " << argument.size() << " elements," << endl;

    while (object.size())
    {
        cout << object.front() << endl;
        object.pop_front();
    }

    return (0);
}
```

Alternatively, `source` may be followed by a iterator of `source`, indicating the first element of `source` that should be spliced, or by two iterators `begin` and `end` defining the iterator-range `[begin, end)` on `source` that should be spliced into `target`.

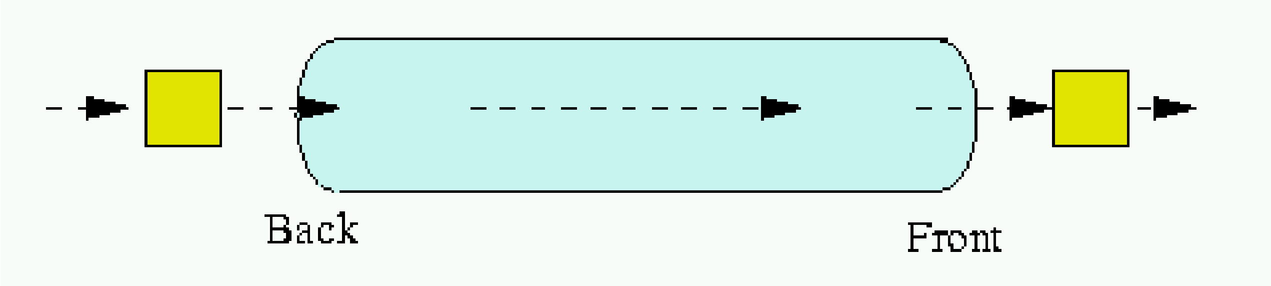
Available operators with the `list` containertype are:

- The assignment of a list to another: `=`,
- The test for equality of two lists: `==`,
- The test for inequality of two lists: `!=`,
- <: This operator returns `true` if each element stored in the left-hand operand list is less than each corresponding element in the right-hand operand list, based on the <-operator of the element-type of the lists. Also available are the `<=`, > and >= operators.

### 8.2.3 The 'queue' container

The `queue` class implements a queue datastructure. To use the `queue`, the header file `queue` must be included:

```
#include <queue>
```




\

Figure 8.4: A queue data-structure

A queue is depicted in figure 8.4.

In figure 8.4 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.

Bearing this model of the queue in mind, let's see what we can do with it.

A queue can be initialized by an existing other queue, or it can be created empty:

```
queue<int>
    queue1;
...
queue<int>
    queue2(queue1);
```

Apart from these constructors, and the basic operators for comparison and assignment (see the introductory paragraph of this chapter), the following member functions are available:

- `empty()`,
- `size()`,
- `front()`: returns the first element that would be removed by `pop()`, Alternatively, the last element of the queue may be reassigned, as illustrated in the following example, in which `Hello World`, rather than `Hello` is displayed:

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

int main()
{
    queue<string>
        q;

    q.push("Hello");
    q.front() = "Hello World";
```

```
    cout << q.front() << endl;

    return (0);
}
```

- `back()`: returns the last element that was added to the queue. Like `front()`, `back()` can be used to reassign the last item that was added to the queue.
- `push(source)`: adds item `source` to the back of the queue.
- `void pop()`: removes (but does not return) the element at the front of the queue.

Note that the queue does not support iterators or a subscript operator. The only elements that can be accessed are its front and back element, and a queue can only be emptied by repeatedly removing its front element.

### 8.2.4 The 'priority_queue' container

The `priority_queue` class implements a priority queue datastructure. To use the priority queue, the header file `queue` must be included:

```
#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 the other passengers.

The priority queue uses the <-operator of the used data type to decide about the priority of the data elements. The smaller the value, the lower the priority. So, the priority queue *could* also be used for sorting values while they arrive.

A simple example of a priority queue application is the following program: it reads words from `cin` and writes a sorted list of words to `cout`:

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

int main()
{
    priority_queue<string>
        q;

    string
        word;

    while (cin >> word)
```

```
        q.push(word);

    while (q.size())
    {
        cout << q.top() << endl;
        q.pop();
    }

    return (0);
}
```

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 for 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 appear first in the queue. Here is the modified program:

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

class Text
{
    public:
        Text(string const &str): s(str)
        {}
        operator string const &() const
        {
            return (s);
        }
        bool operator<(Text const &right) const
        {
            return (s > right.s);
        }
    private:
        string
            s;
};

ostream &operator<<(ostream &ostr, Text const &text)
{
    return (ostr << text);
}

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();
    }
    return (0);
}
```

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. However, the example shows how the priority queue can be fed objects of a special class, in which the `operator<` has been tailored to a particular use.

A priority queue can be initialized by an existing other priority queue, or it can be created empty:

```
    priority_queue<int>
        priority_queue1;
    ...
    priority_queue<int>
        priority_queue2(priority_queue1);
```

Apart from these constructors, and the basic operators for comparison and assignment (see the introductory paragraph of this chapter), the following member functions are available:

- `empty()`,
- `size()`,
- `top()`: returns the first element that would be removed by `pop()`. This element is not removed from the priority queue, and could be given a new value, as in:

  ```
  priority_queue<string>
      pq;
  ...
  pq.top() = "Hello world";
  ```

- `push(argument)`: adds item `argument` to its appropriate position, respecting its priority.

Note that the priority queue does not support iterators or a subscript operator. The only element that can be accessed is its top element, and it can only be emptied by repeatedly removing this element.

### 8.2.5 The 'deque' container

The `deque` class implements a double ended queue (deque) datastructure. To use the `deque` class, the header file `deque` must be included:

```
#include <deque>
```

A *deque* is comparable to a queue, but it allows reading and writing at both ends of the queue. Actually, the `deque` data type supports a lot more functionality than the `queue`, as will be clear from the following overview of member functions that are available for the `deque`:

First, several constructors are available for the `deque`:

- `deque()` initializes an empty deque.
- `deque(argument)` initializes a deque with another deque `argument`.
- `deque(n, argument)` initializes a deque with `n` values provided by the `argument` variable. E.g., to initialize a deque with 10 strings containing `Hello World` we do:

  ```
  deque<string>
      hello(10, "Hello World");
  ```

- `deque(size_type n)` initializes a deque with `n` default values of the datatype stored in the deque.
- `deque(iterator first, iterator last)` initializes the deque with the iterator range implied by `[first, last)`. The iterators `first` and `last` may also be pointers to the data-type stored in the deque.

To access the individual elements of the deque, the following members are available:

- `begin()`: this returns the iterator pointing to the front-element
- `end()`: the iterator beyond the back-element.
- `rbegin()`: the iterator pointing to the last (back) element
- `rend()`: and the corresponding one pointing just before the first (front) element.
- The subscript operator may be used to access random elements from the deque.
- `front()`: returns the element at the front of the deque. This member may be used for reassigning the front element as well.
- `back()`: and the element at the back of the deque. Again, reassignment is possible.
- `size()`, returning the number of elements in the deque.
- `empty()`, returns `true` if the deque contains no elements.

The following operations and operator affect all elements of a deque:

- The assignment operator (=) may be used to assign one deque object to another.
- `swap(argument)` is used to swap the contents of the current deque with deque `argument`.

Elements may be added and removed from both ends of a deque:

- `push_back(source)` adds `source` at the back of the deque,
- `push_front(source)` adds `source` at the front of the deque.
- `pop_back()` removes (but does not return) the element at the back of the deque.
- `pop_front()` removes (but does not return) the element at the front of the deque.

Elements may also inserted somewhere within the deque:

- `insert(position, argument)`: `argument` is inserted at the position indicated by the `position` iterator, which is itself returned by the function. `Argument` may be omitted, in which case the default value of the data-type used with the deque is inserted.
- `insert(pos, n, argument)`: At the position indicated by the `pos` iterator `n` new elements are inserted, all having value `argument`. There is no returnvalue.
- `insert(pos, first, last)`: At the position indicated by the `pos` iterator the elements implied by the iterator range `[first, last)` are inserted. There is no returnvalue.
- `resize(new_size, argument)`: the size of the deque is altered to `new_size`. If `new_size` exceeds `size()`, then the new elements are initialized to `argument`. If `argument` is omitted, the default value of the data type of the deque is used. If `new_size` is less than `size()`, then the size of the deque is merely reduced.

Apart from using `resize()`, elements may be removed from the deque as follows:

- `erase(pos)` erases all elements of the deque from the position indicated by the iterator `pos` to the end of the deque.
- `erase(first, last)` erases all elements implied by the iterator range `[first, last)`.
- `clear()` erases all elements from the deque.

### 8.2.6 The 'map' container

The `map` class implements a (sorted) associative array. To use the `map`, the header file `map` must be included:

```
#include <map>
```

A `map` is filled with *Key/Value* pairs, which may be of any container-acceptable type.

The key is used for looking up the information belonging to the key. The associated information is the *Value*. For example, a phonebook 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.

Basically, the operations on a `map` are the storage of *Key/Value* combinations, and looking for a value, given a key. Each key can be stored only once in a `map`. If the same key is entered twice, the last entered *key/value* pair is stored, and the pair that was entered before is lost.

A single value that must be entered into a `map` must be constructed first. For this, every `map` defines a `value_type` which may be used to create values of that type. For example, a value for a `map<string, int>` can be constructed as follows:

```
map<string, int>::value_type(string("Hello"), 1)
```

The `value_type` is associated with the `map<string, int>` map. Its leftmost argument defines the key, its rightmost argument defines its value.

Instead of using the line `map<string, string>::value_type(...)` over and over again, a `typedef` comes in handy:

```
typedef map<string, int>::value_type MapStrIntValue
```

Using this typedef, values for the `map<string, int>` may be constructed as

```
MapStrIntValue(string("Hello"), 1);
```

Apart from the basic operations (assignment, comparison, etc,), the `map` supports several more operations:

- The constructor defining an empty `map`. The types of the *Key* and *Value* must be specified when the `map` is defined. E.g., to define a `map` in which the key is a `string` and the value an `int`, use:

  ```
  map<string, int>
      object;
  ```

  To define a map in which the key is a `string` and the value is a pair of strings, use:

  ```
  map<string, pair<string, string> >
      object;
  ```

Note the white space between the two closing angular brackets >: this is obligatory, as the immediate concatenation of the two angular brackets will be interpreted by the compiler as a rightshift operator (>>), which is not what you want here.

- `object(iterator first, iterator last)`: This constructor defines a `map` that is initialized by the values implied by the iterator range `[first, last)`. The range could be defined by pointers in an array of *Key/Value* pairs. For example (see section 8.1 for a discussion of the `pair` container):

```
    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]);
```

Note that `&pa[3]`, as with the iterators, points to the first element that must *not* be included in the `map`. The particular array element does not have to exist.

Also note that key/value pairs are only entered if the corresponding key has not yet been entered. 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 ignored silently.

Finally, it is worth noting that the `map` receives its own copies of the data to which the iterators point. The following example illustrates this:

```
#include <iostream>
#include <string>
#include <utility>
#include <map>

class MyClass
{
    public:
        MyClass()
        {
            cout << "MyClass constructor called\n";
        }
        MyClass(const MyClass &other)
        {
            cout << "MyClass copy constructor called\n";
        }
        ~MyClass()
        {
            cout << "MyClass destructor called\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";

    return (0);
}
```

First, the constructors of a `MyClass` object is called to initialize the first element of the array `pairs`. This object is copied into the first element of the array `pairs` by calling the copy constructor. Next, the original element is not needed anymore, and gets destroyed. At that point the array `pairs` is constructed. Next, the `map` constructs a temporary `pair` object, from which the map element is constructed. 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.

When run, the program produces the following output:

```
MyClass constructor called
MyClass copy constructor called
MyClass destructor called
pairs constructed
MyClass copy constructor called
MyClass copy constructor called
MyClass destructor called
mapsm constructed
MyClass destructor called
```

- `object(argument)`: This constructor initializes `object` with an existing map `argument` having the same key/value combinations.

The standard iterators are also available:

- `begin()`
- `end()`
- `rbegin()`
- `rend()`

Other member functions of the `map` are:

- `empty()`,
- `size()`,
- `swap()`,
- The subscript operator (`[]`), which may be used to access and redefine values. Here, the argument of the subscript operator is the keyvalue. If the provided key is not available in the `map`, a new data element is added to the `map`, using the default value or default constructor to initialize the value part of the newly added key/value combination. This default value is then returned.

  When initializing a new or reassigning another element of the map, the right-hand side of the assignment operator must have the type of the value part of the map. E.g., to add another element `"two"` to the map that was defined in the previous example, use the following construction:

  ```
      mapsm["two"] = MyClass();
  ```

- `insert(argument)` is used to insert a new value `argument` in the map. The returnvalue is a `pair<iterator,bool>`. The `bool` field indicates whether `source` was inserted (`true` is returned) or not (in which case the `key` field of `source` was already available). In both cases the iterator field points to the data-element in the `map`: a new element if `true` is returned, the existing element if `false` is returned. The following little program illustrates this:

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

  int main()
  {
      pair<string, int>
          pa[] = {
              pair<string,int>("one", 1),
              pair<string,int>("two", 2),
              pair<string,int>("three", 3),
          };

      map<string, int>
          xmap(&pa[0], &pa[3]);

                              // {four, 4} and true (1) is returned  here
      pair<map<string, int>::iterator, bool>
          ret = xmap.insert(map<string, int>::value_type("four", 4));

      cout << ret.first->first << " " << ret.first->second << " " <<
              ret.second << " " << xmap["four"] << endl;

                              // {four, 4} and false (0) is returned  here
      ret = xmap.insert(map<string, int>::value_type("four", 0));

      cout << ret.first->first << " " << ret.first->second << " " <<
              ret.second << " " << xmap["four"] << endl;
  ```

```
    return (0);
}
```

Note the somewhat peculiar constructions like

```
    cout << ret.first->first << " " << ret.first->second << ...
```

Here `ret` is the `pair` variable returned by the `insert` member function. Its `first` field is an iterator into the t(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* field of the map value (a `string`), and `ret.first->second` is the *value* field (an `int`).

- `insert(position, argument)`. This is another way to insert a value, this time using a specific position within the `map`. `Position` is an `map<keytype, valuetype>::iterator`. Although a specific position is given, the new element is inserted at its appropriate sorted location within the `map`, so `mapVariable.begin()` could be used for the `position`.
- `insert(first, last)`: this member function may be used to insert a range of elements implied by the iterator range `[first, last)` into the `map`. Again, elements are only inserted if their keys are not yet in the `map`, and the `map` remains sorted by key values. Instead of iterators *pointers* to elements of the same value type as stored in the `map` may be used.
- `erase(position)`: erases the element at the indicated position, which is an iterator of the particular map.
- `erase(key)`: erases the element having `key` as its key value.
- `erase(first, last)`: erases the range of elements implied by the iterator range `[first, last)`.
- `clear()`: erases all elements from the `map`.
- `find(key)`: an iterator is returned pointing to the element whose key is `key`. If the element isn't available, the iterator `end()` is returned. The following example illustrates the use of the `find()` member function:

```
#include <iostream>
#include <string>
#include <utility>
#include <map>

int main()
{
    map<string, int>
        mapsi;

    mapsi["one"] = 1;

    map<string, int>::iterator
        it = mapsi.find("one");
```

```
        cout << "\"one\" " <<
                (it == mapsi.end() ? "not " : "") <<
                "found\n";

        it = mapsi.find("three");

        cout << "\"three\" " <<
                (it == mapsi.end() ? "not " : "") <<
                "found\n";

        return (0);
    }
```

The following members have special meanings with the multimap, but they are defined with the plain `map` too:

- `count(key)`: returns 1 if the provided key is available in the `map`, otherwise 0 is returned.
- `lower_bound(key)`: returns an iterator pointing to the first element having a key equal to or exceeding the key value that is passed to the member function. If no such value exists, `target.end()` is returned.
- `upper_bound(key_type key)`: same as the previous function.
- `equal_range(key_type key)`: a `pair<iterator,iterator>` is returned. In the case of a `map`, the range consists of the data element having as its key the key value that is passed to the function. If no such data element could be found, the pair `(end(), end())` is returned.

### 8.2.7 The 'multimap' container

Like the `map`, the `multimap` class implements also a (sorted) associative array. To use the `multimap`, the header file `multimap` must be included:

```
#include <multimap>
```

The main difference between the `map` and the `multimap` is that the multimap supports multiple entries of the same key, whereas the `map` contains only unique keys. Note that multiple entries of the same key *and* the same value are also accepted.

The functions that are available with the multimap and the map are practically the same, with the exception of the subscript 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 `myMap[myKey]`?

Below the available constructors and member functions are mentioned. They are presented without further comment if their function is identical to that of the `map` container.

A single value that is to be entered into a multimap must be constructed. For this, a multimap defines a `value_type`, corresponding to a particular multimap type, which may be used to create values of that type. For example, with a `multimap<string, string>` it can be used as follows:

```
multimap<string, string>::value_type(string("Hello"), 1)
```

Here are the constructors that are available for the multimap:

- The constructor defining an empty multimap. E.g.,

```
multimap<string, int>
    object;
```

- `object(first, last)`: This constructor defines a multimap that is initialized by the values implied by the iterator range `[first, last)`.
- `object(argument)`: This constructor initializes `object` with an existing multimap.

The standard iterator producing member functions are available:

- `begin()`
- `end()`
- `rbegin()`
- `rend()`

Other available member functions are:

- `empty()`
- `size()`
- `swap()`
- `insert(argument)` is used to insert a new value `argument` in the multimap. The returnvalue is an *iterator* (and *not* a `pair<iterator,bool>` as with the map container), pointing to the newly added element.
- `insert(position, argument)`.
- `insert(first, last)`.
- `erase(position)`.
- `erase(key)`.
- `erase(first, last)`.
- `clear()`.
- `find(key)`: an iterator is returned pointing to the (first) element whose key is `key`. If the element isn't available, `target.end()` is returned.
- `count(key)`: returns the number of times the provided key is available in the multimap.

- `lower_bound(key)`: returns an iterator pointing to the first of a series of data element having the same keys of which the value is equal to or exceeds the key value that is passed to the member function. If no such value exists, the behavior of the function is undefined.
- `upper_bound(key)`: returns an iterator pointing to the last of a series of data element having the same keys of which the value is equal to or exceeds the key value that is passed to the member function. If no such value exists, the behavior of the function is undefined.
- `equal_range(key)`: a `pair<iterator,iterator>` is returned, defining the range of data elements all having key value `key`. If no such data element could be found, the pair `(end(), end())` is returned.

The subscript operator is *not* available.

### 8.2.8 The 'set' container

The `set` class implements a set of (sorted) values. To use the `set`, the header file `set` must be included:

```
#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 single value that is to be entered in a set must be constructed. For this, a set defines a `value_type`, corresponding to a particular type of set, which may be used to create values of that type. For example, with a `set<string>` it can be used as follows:

```
set<string>::value_type(string("Hello"))
```

Instead of using the line `set<string>::value_type(...)` over and over again, a `typedef` may come in handy here:

```
typedef set<string>::value_type SetSValue
```

Using this typedef, values for the `set<string>` may be constructed as follows:

```
SetSValue(string("Hello"))
```

Apart from the basic operations (assignment, comparison, etc,), the `set` supports several more operations. They are:

- The constructor defining an empty set. When the set is defined, the type of the value must be specified. E.g., to define a set in which `int`s can be stored, use:

```
set<int>
    object;
```

- `object(iterator first, iterator last)`: This constructor defines a set that is initialized by the values implied by the iterator range `[first, last)`. The range may also be defined by pointers in an array of values of the same type as the values that must be stored in the set. For example:

```
int
    ia[] = {1, 2, 3, 4, 5};

set<int>
    object(&ia[0], &ia[5]);
```

  Note that `&ia[5]` points to the first element that must *not* be included in the set. Also note that all values in the set will be different: it is not possible to store the same value more than once.

- `object(argument)`: This constructor initializes `object` with an existing set `argument`, constructing a copy of the set `argument`.

The standard iterators are all available:

- `begin()`
- `end()`
- `rbegin()`
- `rend()`

Other member functions are:

- `empty()`,
- `size()`,
- `swap(argument)`, swapping the contents of the current set and the set `argument`.
- `insert(argument)` is used to insert a new value `argument` in the set. `Argument` is a value of the appropriate value type of the set. The returnvalue is a `pair`<`iterator, bool`>. The `bool` field indicates whether `source` was inserted (`true` is returned) or not (in which case the `key` field of `source` was already available). In both cases the iterator field points to the data-element in the set: a new element if `true` is returned, the existing element if `false` is returned. An example using the `insert()` member function is given below:

```
#include <set>
#include <utility>

int main()
{
    set<int>
        object;

    pair<set<int>::iterator, bool>
        result = object.insert(set<int>::value_type(4));

    cout << "Element " << *result.first << " was " <<
        (result.second ? "" : "not ") << "inserted\n";

    result = object.insert(set<int>::value_type(4));

    cout << "Element " << *result.first << " was " <<
        (result.second ? "" : "not ") << "inserted\n";

    return (0);
}
```

- `insert(position, argument)`. This is another way to insert a value `argument`, this time using a specific position within the set, indicated by `set<type>::iterator position`. Although a specific position is given, the new element is inserted at its appropriate sorted location within the set. An insertion could therefore be realized using a statement like

```
    object.insert(object.begin(), set<int>::value_type(1));
```

- `insert(first, last)`: this member function may be used to insert a range of elements implied by the iterator range `[first, last)` into the set. Again, elements are only inserted if their keys are not yet in the set, and the set remains sorted. Instead of iterators *pointers* to elements of the same value type as stored in the set may be used.
- `erase(position)`: erases the element at the indicated `set<type>::iterator position`.
- `erase(argument)`: erases the element having `argument` as its value.
- `erase(first, last)`: erases the range of elements implied by the iterator range `[first, last)`.
- `clear()`: erases all elements from the set.
- `find(argument)`: an iterator is returned pointing to the element whose value is `argument`. If the element isn't available, `object.end()` is returned.

The following members have special meanings with the multiset, but they are defined with the plain set too:

- `count(argument)`: returns 1 if the provided value is available in the set, otherwise 0 is returned.

- `lower_bound(argument)`: returns an iterator pointing to the (first) data element having a value which is equal to or exceeds the value that is passed to the member function. If no such value exists, the behavior of the function is undefined.
- `upper_bound(argument)`: same as the previous function.
- `equal_range(argument)`: a `pair<set<type>::iterator, set<type>::iterator>` is returned. In the case of a set, the range consists of a pair of iterators of which the first iterator points to the element of the set containing the value `argument`, while the second iterator points beyond that element (or to `end()` if the first iterator points to the last element in the set). If the set does not contain a data element having value `argument` the pair `(end(), end())` is returned.

### 8.2.9 The 'multiset' container

Like the `set`, the `multiset` class also implements a (sorted) set of values. To use the `multiset`, the header file `multiset` must be included:

```
#include <multiset>
```

The main difference between the `set` and the `multiset` is that the multiset supports multiple entries of the same value, whereas the `set` contains only unique values.

The member functions that are available for the set are also available for the multiset. They are presented below without further comment if their functions and parameters are comparable to those used by the `set` container's members.

A single value that is to be entered into a multiset must be constructed. For this, a multiset defines a `value_type`, corresponding to a particular multiset type, which may be used to create values of that type. For example, with a `multiset<string>` it can be used as follows:

```
multiset<string>::value_type(string("Hello"))
```

Here are the constructors that are available for the multiset:

- The constructor defining an empty multiset. E.g.,

  ```
  multiset<string>
      object;
  ```

- `object(first, last)`: This constructor defines a multiset that is initialized by the values implied by the iterator range `[first, last)`.
- `object(argument)`: This constructor initializes `object` with an existing multiset `argument`, creating a copy of that multiset.

The standard iterators:

- begin()
- end()
- rbegin()
- rend()

Other member functions are:

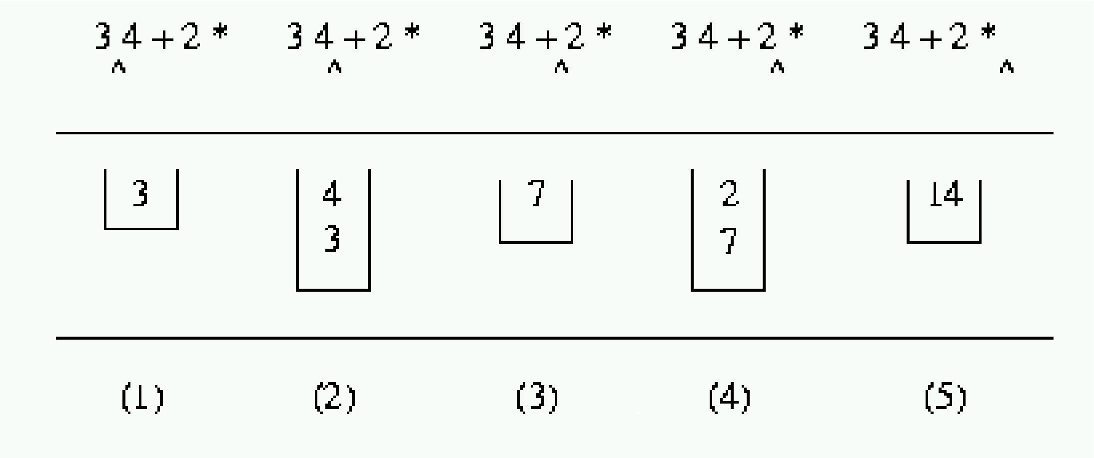
- empty(),
- size(),
- swap(argument), argument is an existing multiset.
- insert(argument) is used to insert a new value multiset<type>::value_type(argument) into the multiset. The returnvalue is an *iterator* (and *not* a pair<iterator, bool> as with the set container), pointing to the newly added element.
- insert(position, argument). Position is an iterator of the multiset, and argument is a value for the multiset.
- insert(first, last), inserting values defined by the iterator range [first, last).
- erase(position),
- erase(argument),
- erase(first, last),
- clear().
- find(argument): an iterator is returned pointing to the (first) element whose value is argument. If the element isn't available, object.end() is returned.
- count(argument): returns the number of times the provided value argument is available in the multiset.
- lower_bound(argument): returns an iterator pointing to the first of a series of data element having values which are equal to or exceed the value argument that is passed to the member function. If no such value exists, the behavior of the function is undefined.
- upper_bound(value): returns an iterator pointing to the last of a series of data element having values which are equal to or exceed the value argument that is passed to the member function. If no such value exists, the behavior of the function is undefined.
- equal_range(argument): a pair<iterator, iterator> is returned, defining the range of data elements all having the value argument. If no such elements could be found, the pair (end(), end()) is returned.

A small example showing the use of various member functions of a multiset is given below:

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

int main()
{
    string
        sa[] =
        {
            "alpha",
            "echo",
            "hotel",
            "mike",
            "romeo"
        };

    multiset<string>
        xset(&sa[0], &sa[5]);

    xset.insert(multiset<string> ::value_type("echo"));
    xset.insert(multiset<string> ::value_type("echo"));
    xset.insert(multiset<string> ::value_type("echo"));

    multiset<string>::iterator
        it = xset.find("echo");

    for (; it != xset.end(); ++it)
        cout << *it << " ";
    cout << endl;

    pair
    <
        multiset<string>::iterator,
        multiset<string>::iterator
    >
        itpair = xset.equal_range("echo");

    for (; itpair.first != itpair.second; ++itpair.first)
        cout << *itpair.first << " ";

    cout << endl <<
            xset.count("echo") << " occurrences of 'echo'" << endl;


    return (0);
}
```

### 8.2.10 The 'stack' container

The `stack` class implements a stack datastructure. To use the `stack`, the header file `stack` must be included:




\

Figure 8.5: The contents of a stack while evaluating `3 4 + 2 *`

```
#include <stack>
```

A stack is also called a *first-in last-out* datastructure, as the first item to enter the stack is the last item that will be removed from it. A stack is an extremely useful datastructure in situations where data must be temporarily be available. For example, programs maintain a stack to store local variables of functions: these variables live only as long as the functions live, contrary to global (or static local) variables, which live for as long as the program itself lives. Another example is found in calculators using the *Reverse Polish Notation (RPN)*, in which the operands of expressions are entered in the stack, and the operators pop their operands and push the results of their work.

As an example of the use of a stack, consider figure 8.5, 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 8.5 shows the stack contents after each *token* (i.e., the operands and the operators) is read from the input. Notice that indeed each operand is 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 8.5 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 8.5 we see that a stack has one point (the *top*) where items can be added to and removed from the stack. Furthermore, values can be pushed and popped from a stack.

Bearing this model of the stack in mind, let's see what we can formally do with it, using the `stack` container.

A stack can be initialized by an existing other stack, or it can be created empty:

```
stack<int>
    stack1;
...
stack<int>
    stack2(stack1);
```

Apart from these constructors, and the basic operators for comparison and assignment (see the introductory paragraph of this chapter), the following member functions are available:

- `empty()`,
- `size()`,
- `top()`: returns the first element that would be removed by `pop()`. Using `top()` the value at the top of the stack may be inspected or reassigned.
- `push(argument)`: pushes item `argument` on the stack.
- `void pop()`: removes (but does not return) the element at the top of the stack.

Note that the stack does not support iterators or a subscript operator. The only elements that can be accessed is its top element, and it can only be emptied by repeatedly popping the element at the top.

### 8.2.11 The 'hash_map' and other hashing-based containers

The `(multi) map` and `(multi) set` containertypes store sorted keys. This is in general not the fastest way to store keys with respect to storage and retrieval. The main benefit of sorted keys is that a listing of sorted keys appeals more to humans than an unsorted list. However, a by far faster method of storing keys is to use *hashing*.

Hashing uses a function (called the *hash*-function) to compute a (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 hashvalue of the provided key, and looking at the table in the computed indexlocation: 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.

Boundary conditions arise 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 illustrated. The 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 hashvalue for the key, and an object comparing two keys for equality.

The `hash_map` class implements an associative array in which the key is stored according to some hashing scheme. To use the `hash_map`, the header file `hash_map` must be included:

```
#include <hash_map>
```

Hash functions are available for `char const *` keys, and for all the scalar numerical types `char, short, int` etc.. If another datatype must be used, a hash function and an equality test must be implemented, possibly using *function objects* (see section 7.8). For both situations examples are given below.

The class implementing the hash-function could be called `hash`. Its function-call operator returns the hashvalue of the key which is passed as its argument.

A *generic algorithm* (see section 11) 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 function object could also be constructed here, supporting the equality test of two keys. Again, both situations are illustrated below.

In the first example a `hash_map` is defined for a `string, int` combination using existing template functions.

The test for equality is implemented using an instantiation of the `equal_to` generic algorithm. The hash function uses a *template specialization* for the `hash` template class. The how and why of template specializations are covered in chapter 17.

The `hash<string>` explicit specialization in fact uses the predefined `hash<char const *>` template, but the roundabout way is chosen here to illustrate how a template explicit specialization can be constructed. Here it is:

```
template <>
class hash<string>
{
    public:
        size_t operator()(string const &str) const
        {
            hash<char const *>
                h;

            return (h(str.c_str()));
        }
};
```

The following program defines a 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>
#include <string>
#include <hash_map>
```

```
template <> class hash<string>; // insert the above mentioned template
                                // here

int main()
{
    hash_map<string, int, hash<string> >
        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;

    return (0);
}
```

Note that the definition `hash_map<string, int, hash<string> > months;` may be written simpler if the key is a `char const *`: `hash_map<char const *, int> months;`

The next example shows an alternative implementation, using function objects. The class `Equal` defines the equality test of two keys in its function call operator `operator()`, and a `Equal` object is now explicitly mentioned when the `hash_map` is constructed. Similarly, the `hashString` class defines the hash function of the key. A `hashString` object is also passed explicitly to the constructor of the `hash_map`:

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

class Equal
{
    public:
        size_t operator()(string const &s1, string const &s2) const
        {
            return (s1 == s2);
        }
};

class hashString
```

```
{
    public:
        size_t operator()(string const &str) const
        {
            hash<char const *>
                h;

            return (h(str.c_str()));
        }
};

int main()
{
    hash_map
    <
        string,
        int,
        hashString,
        Equal
    >
        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 << "february -> " << months["february"] << endl <<
            "april     -> " << months["april"] << endl <<
            "june      -> " << months["june"] << endl <<
            "november  -> " << months["november"] << endl <<
            "december  -> " << months["december"] << endl;

    return (0);
}
```

Like the `map`, a single value that will be entered into a hash_map must be constructed. For this, a `hash_map` defines a `value_type`, corresponding to a particular `hash_map`-type, which may be used to create values of that type. For example, with a `hash_map<string, int>` it can be used as follows:

```
hash_map<string, int>::value_type(string("Hello"), 1)
```

All the member functions and constructors that are available for the map datatype can also be used for the hash_map. The constructor object(n) defines a hash_map consisting of an initial number of n slots to put key/value combinations in. This number is automatically extended when needed.

The hash_multimap, hash_set and hash_multiset containers are used analogously. For these containers the equal and hash classes must also be defined. The hash_multimap also requires the hash_map header file, the hash_set and hash_multiset containers can be used after including the hash_set header file. Be careful not to use the subscript operator with the hash_multimap and hash_multiset, as this operator is not defined for the multi_... containers.

## 8.3 The 'complex' container

The complex container is a specialized container in that it defines operations that can be performed on complex numbers, given possible numerical real and imaginary data types.

In order to use the complex container, the headerfile

```
#include <complex>
```

must be included.

The complex container can be used to define complex numbers, consisting of two parts, representing the real and complex 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 typedefinition requires the specification of the datatype of the real and imaginary parts. E.g.,

```
complex<double>
complex<int>
complex<float>
```

Note that the real and imaginary parts of complex numbers have the same datatypes.

Below it is silently assumed that the used complex type is complex<double>. Given this assumption, complex numbers may be initialized as follows:

- target: A default initialization: real and imaginary parts are 0.
- target(1): The real part is 1, imaginary part is 0
- target(0, 3.5): The real part is 0, imaginary part is 3.5
- target(source): target is initialized with the values of source.

Anonymous complex values may also be used. In the following example two anonymous complex values are pushed on a stack of complex numbers, to be popped again thereafter:

```
#include <iostream>
#include <complex>
#include <stack>

int main()
{
    stack<complex<double> >
        cstack;

    cstack.push(complex<double>(3.14, 2.71));
    cstack.push(complex<double>(-3.14, -2.71));

    while (cstack.size())
    {
        cout << cstack.top().real() << ", " <<
                cstack.top().imag() << "i" << endl;
        cstack.pop();
    }

    return (0);
}
```

Note that a blank is required between the two consecutive >-barckets used in the definition of `cstack`. If the blank is omitted, the resulting >> is read as the right-shift operator, which of course makes no sense here.

The following member functions and operators are defined for complex numbers:

- The standard assignment and comparison operators that are available for containers are also available for complex numbers.
- `real()`: this member function returns the real part of a complex number.
- `imag()`: this member function returns the imaginary part of a complex number.
- The following operations are defined for `complex` containers: `+, -, *, /, +=, -=, *=, /=`.

Furthermore, several mathematical functions are available for complex numbers. They are `abs()`, arg(), conj(), cos(), cosh(), exp(), log(), norm(), polar(), pow(), sin(), sinh()) and `sqrt()`. These functions are normal functions, not member functions. They accept complex numbers as their arguments. For example,

```
    abs(complex<double>(3, -5));
    pow(target, complex<int>(2, 3));
```

Complex numbers may be extracted from `istream` objects and inserted into `ostream` objects. The insertion results in an ordered pair `(x, y)`, in which `x` represents the real part and `y` the imaginary part of the complex number. The same form may also be used when extracting a complex number from an `istream` object. However, simpler forms are also allowed. E.g., `1.2345`: only the real part, the imaginary part will be set to 0; `(1.2345)`: the same value.

Finally, ordinary numbers may be used in expressions involving complex numbers. E.g.,

```
// assume target is complex<double>:
target *= 3;
```

Note, however, that the reverse does not hold true: a complex number cannot be assigned to a non-complex type variable. In these situations the `real()`, `imag()` or other functions must be used. E.g.:

```
// assume x is double:
x = target;         // error: x is not complex<double>
x = target.real();  // ok.
```

# Chapter 9

# Static data and functions

In the previous chapters we have shown examples of classes where each object of a class had its own set of `public` or `private` data. Each `public` or `private` function could access the object's own version of the data.

In some situations it may be desirable that one or more *common* data fields exist, which are accessible to *all* objects of the class. An example of such a situation is the name of the startup directory in a program which recursively scans the directory tree of a disk. A second example is a flag variable, which states whether some specific initialization has occurred: only the first object of the class would then perform the initialization and would then set the flag to 'done'.

Such situations are analogous to **C** code, where several functions need to access the same variable. A common solution in **C** is to define all these functions in one source file and to declare the variable as a `static`: the variable name is then not known beyond the scope of the source file. This approach is quite valid, but doesn't stroke with our philosophy of one function per source file. Another **C**-solution is to give the variable in question an unusual name, e.g., `_6uldv8`, and then to hope that other program parts won't use this name by accident. Neither the first, nor the second **C**-like solution is elegant.

**C++**'s solution is to define `static` data and functions, common to all objects of a class, and inaccessible outside of the class. These functions and data will be discussed in this chapter.

## 9.1 Static data

A data member of a class can be declared `static`; be it in the `public` or `private` part of the class definition. Such a data member is created and initialized only once, in contrast to non-static data members, which are created again and again, for each separate object of the class. A `static` data member is created once: when the program starts executing. Nonetheless, it is still part of the class.

`static` data members which are declared `public` are like 'normal' global variables: they can be reached by *all code of the program* using their name, together with their class name and the scope resolution operator. This is illustrated in the following code fragment:

```
class Test
```

```
{
    public:
        static int
            public_int;
    private:
        static int
            private_int;
}

int main()
{
    Test::public_int = 145;     // ok

    Test::private_int = 12;     // wrong, don't touch
                                // the private parts
    return (0);
}
```

This code fragment is not suitable for consumption by a **C++** compiler: it only illustrates the *interface*, and not the *implementation* of `static` data members. We will discuss the implementation of such members shortly.

### 9.1.1 Private static data

To illustrate the use of a `static` data member which is a `private` variable in a class, consider the following code fragment:

```
class Directory
{
    public:
        // constructors, destructors, etc. (not shown)
        ...
    private:
        // data members
        static char
            path[];
};
```

The data member `path[]` is a `private static` variable. During the execution of the program, only *one* `Directory::path[]` exists, even though more than one object of the class `Directory` may exist. This data member could be inspected or altered by the constructor, destructor or by any other member function of the class `Directory`.

Since constructors are called for each new object of a class, `static` data members are never *initialized* by constructors. At most they are *modified*. The reason for this is that the `static` data members exist *before* the constructor of the class is called for the very first time. The `static` data members can be initialized during their definition, outside of all member functions, in the same way as global variables are initialized. The definition and initialization of a `static` data member usually occurs in one of the source files of the class functions, preferably in a source file dedicated to the definition of static data members, called `data.cc`.

The data member `path[]` from the above class `Directory` could thus be defined and initialized in the source file of the constructor (or in a separate file `data.cc`):

```
// the static data member: definition and initialization
char
    Directory::path [200] = "/usr/local";

// the default constructor
Directory::Directory()
{
    ...
}
```

It should be noted that the definition of the `static` data member can occur in *any* source file; as long as it is defined only once. So, there is no need to define it in, e.g., a source file in which also a member function of the class is implemented.

In the class interface the `static` member is actually only *declared*. At its implementation (definition) its type and class name are explicitly stated. Note also that the size specification can be left out of the interface, as is shown in the above array `path[]`. However, its size *is* needed at its implementation.

A second example of a useful `private static` data member is given below. A class `Graphics` defines the communication of a program with a graphics-capable device (e.g., a VGA screen). The initial preparing of the device, which in this case would be to switch from text mode to graphics mode, is an action of the constructor and depends on a `static` flag variable `nobjects`. The variable `nobjects` simply counts the number of `Graphics` objects which are present at one time. Similarly, the destructor of the class may switch back from graphics mode to text mode when the last `Graphics` object ceases to exist.

The class interface for this `Graphics` class might be:

```
class Graphics
{
    public:
        // constructor, destructor
        Graphics();
        ~Graphics();

        // other interface is not shown here,
        // e.g. to draw lines or whatever

    private:
        // counter of # of objects
        static int nobjects;

        // hypothetical functions to switch to graphics
```

```
        // mode or back to text mode
        void setgraphicsmode();
        void settextmode();
}
```

The purpose of the variable `nobjects` is to count the number of objects which exist at one given time. When the first object is created, the graphics device is initialized. At the destruction of the last `Graphics` object, the switch from graphics mode to text mode is made:

```
// the static data member
int Graphics::nobjects = 0;

// the constructor
Graphics::Graphics()
{
    if (! nobjects)
        setgraphicsmode();
    nobjects++;
}

// the destructor
Graphics::~Graphics()
{
    nobjects--;
    if (! nobjects)
        settextmode();
}
```

It is obvious that when the class `Graphics` would define more than one constructor, each constructor would need to increase the variable `nobjects` and possibly would have to initialize the graphics mode.

### 9.1.2 Public static data

Data members can be declared in the `public` section of a class definition, although this is not common practice (such a setup would violate the principle of data hiding). E.g., when the `static` data member `path[]` from chapter 9.1 would be declared in the `public` section of the class definition, all program code could access this variable:

```
int main()
{
    getcwd(Directory::path, 199);
    return(0);
}
```

Note that the variable path would still have to be defined. As before, the class interface would only *declare* the array path[]. This means that some source file would still need to contain the implementation:

```
char
    Directory::path[200];
```

## 9.2 Static member functions

Besides static data, C++ allows the definition of static functions. Similar to the concept of static data, in which these variables are shared by all objects of the class, static functions apply to all objects of the class.

The static functions can therefore address only the static data of a class; non-static data are unavailable to these functions. If non-static data could be addressed, to which object would they belong? Similarly, static functions cannot call non-static functions of the class. All this is caused by the fact that static functions have no this pointer.

Functions which are static and which are declared in the public section of a class interface can be called without specifying an object of the class. This is illustrated in the following code fragment:

```
class Directory
{
    public:
        // constructors, destructors etc. not shown here
        ...
        // here's the static public function
        static void setpath(char const *newpath);

    private:
        // the static string
        static char path [];
};

// implementation of the static variable
char Directory::path [199] = "/usr/local";

// the static function
void Directory::setpath(char const *newpath)
{
    strncpy(path, newpath, 199);
}

// example of the usage
int main()
{
    // Alternative (1): calling setpath() without
    // an object of the class Directory
```

```
        Directory::setpath("/etc");

        // Alternative (2): with an object
        Directory
            dir;

        dir.setpath("/etc");

        return (0);
    }
```

In the example above the function `setpath()` is a `public static` function. **C++** also allows `private static` functions: these functions can only be called from other member functions of the class of which they are themselves members, but not from other functions.

Note that such a `private static` function could only (a) access static variables, or (b) call other static functions: non-static code or data members would still be inaccessible to the static function.

# Chapter 10

# Classes having pointers to members

Pointers in classes have been discussed in detail in chapter 6.1. 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 free 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. The realization of pointers to *members* of an object is the subject of this part of the **C++** annotations.

## 10.1 Pointers to members: an example

Knowing how pointers to variables and objects are to be used does not intuitively lead to the concept of pointer to *members*. Even if the returntype and parametertypes of a member function are taken into account, surprises are encountered. For example, consider the following class:

```
class String
{
    public:
        ...
        char const *get() const;
    private:
        ...
        char const *(*sp)() const;
};
```

Within this class, it is not possible to define a `char const *(*sp)() const` pointing to the `get()` member function of the `String` class.

One of the reasons why this doesn't work is that the variable `sp` has a global scope, while the member function `get()` is defined within the `String` class. The fact that the variable `sp` is part of the `String` class is of no relevance. According to `sp`'s definition, it points to a function outside of the class.

Consequently, in order to define a pointer to a member (either data or function, but usually a function) of a class, the scope of the pointer must be within the class' scope. Doing so, a pointer to a member of the class `String` can be defined as

```
char const
    *(String::*sp)() const;
```

So, due to the `String::` prefix, `sp` is defined to be active only in the context of the class `String`. In this context, it is defined as a pointer to a const function, not expecting arguments, and returning a pointer to const chars.

## 10.2 Initializing pointers to members

Pointers to members can be initialized to point to intended members. Such a pointer can be defined either inside or outside a member function.

Initializing or assigning an address to such a pointer does nothing but indicating which member the pointer will point to. However, member functions (except for the static member functions) can only be used when associated with an object of the member function's class. The same holds true for pointers to data members.

While it is allowed to *initialize* such a pointer outside of the class, it is not possible to *access* such a function without an associated object.

In the following example these characteristics are illustrated. First, a pointer is initialized to point to the function `String::get()`. In this case *no* `String` object is required.

Next, a `String` object is defined, and the string that is stored within the object is retrieved through the pointer, and not directly by the function `String::get()`. Note that the pointer is a variable existing outside of the class' context. This presents no problem, as the actual object to be used is identified by the statement in which object and pointervariable are combined. Consider the following piece of code:

```
void fun()
{
    char const
        *(String::*sp)() const;

    sp = String::get;   // assign the address
                        // of String's get()
                        // function
```

```
    String              // define a String object
        s("Hello world");

    cout << (s.*sp)()   // show the string
         << endl;

    String
        *ps;            // pointer to a String object

    ps = &s;            // initialize ps to point at s

    cout << (ps->*sp)() // show the string again
         << endl;
}
```

Note in this example the statement `(s.*sp)()`. The `.*` construction indicates that `sp` is a pointer to a member function. Since the pointer variable `sp` points to the `String::get()` function, this function is now called, producing the string "`Hello world`".

Furthermore, note the parentheses around `(s.*sp)`. These parentheses are required. If they were omitted, then the default interpretation (now parenthesized for further emphasis) would be `s.* (sp())`. This latter construction means

- Call function `sp()`, which should return a pointer to a member. E.g., `sp()` has the prototype

  ```
  char const * (String::*)() sp();
  ```

  So, `sp()` is a function returning a pointer to a member function of the class `String`, while such a member function must return a pointer to `const chars`.

- Apply this pointer with regard to object s.

Not an impossible or unlikely construction, but wrong as far as the current definition of `sp` is concerned.

When a pointer to a member function is associated with an object, the pointer to member selector operator `.*` is used. When a pointer to an object is used (instead of the object itself) the "pointer to member through a pointer to a class object" operator `->*` operator is required. The use of this operator is also illustrated in the above example.

## 10.3 Pointers to static members

Static members of a class exist without an object of their class. In other words, they can exist *outside of* any object of their class.

When these static members are public, they can be accessed in a 'stand-alone' fashion.

Assume that the `String` class also 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;
}
```

Since pointers to members are always associated with an object, the use of a pointer to a member function would normally produce an error. However, *static* members are actually *global variables or functions*, bound to their class.

Public static members can be treated as globally accessible functions and data. Private static members, on the other hand, can be accessed only from within the context of their class: they can only be accessed from inside the member functions of their class.

Since static members have no particular link with objects of their class, but look a lot like global functions, a pointer variable that is *not* part of the class of the member function must be used.

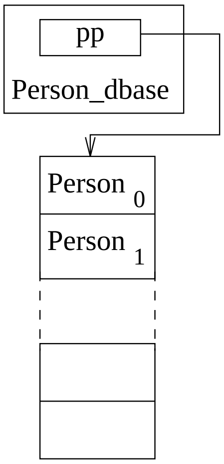
Consequently, a variable `int (*pfi)()` can be used to point to the static member function `int String::n_strings()`, even though `int (*pfi)()` has *nothing* in common with the `class String`. This is illustrated in the next example:

```
void fun()
{
    int
        (*pfi)();

    pfi = String::n_strings;
                // address of the  static member function

    cout << pfi() << endl;
                // print the value produced by
                // String::n_strings()
}
```

## 10.4 Using pointers to members for real

Let's assume that a database is created in which information about persons is stored. Name, street names, city names, house numbers, birthdays, etc. are collected in objects of the class `Person`, which are, in turn, stored in a class `Person_dbase`. Partial interfaces of these classes could be designed as follows:

```
class Date;

class Person()
{
    public:
        ...
```




\

Figure 10.1: Person_dbase objects: Persons reached via Person *pp

```
        char const *get_name() const;
        Date const &birthdate() const;
    private:
        ...
};

class Person_dbase
{
    public:
        enum Listtype
        {
            list_by_name,
            list_by_birthday,
        };
        void list(Listtype type);
    private:
        Person
            *pp;    // pointer to the info
        unsigned
            n;      // number of persons stored.
};
```

The organization of `Person` and `Person_dbase` is pictured in figure 10.1: Within a `Person_dbase` object the `Person` objects are stored. They can be reached via the pointer variable `Person *pp`.

We would like to develop the function `Person_dbase::list()` in such a way that it lists the contents of the database sorted according to a selected field of a `Person` object.

So, when `list()` is called to list the database sorted by *names*, the database of `Person` objects is first sorted by names, and is then listed.

Alternatively, when `list()` is called to list the database sorted by *birthdates*, the database of `Person` objects is first sorted by birthdates, and is then listed.

In this situation, the function `qsort()` is most likely called to do the actual sorting of the `Person` objects[1]. This function requires a pointer to a compare function, comparing two elements of the array to be sorted. The prototype of this compare function is

```
int (*)(void const *, void const *)
```

However, when used with `Person` objects, the prototype of the `compare()` function should be

```
int (*)(Person const *, Person const *)
```

Somewhere a typecast will be required: either when calling `qsort()`, or within the `compare()` functions themselves. We will use the typecast when calling `qsort()`, using the following typedef to reduce the verbosity of the typecasts (a *pointer to an integer function requiring two void pointers*):

```
typedef int (*pif2vp)(void const *, void const *)
```

Next, the function `list()` could be developed according to the following setup:

```
void Person_dbase::list(Listtype type)
{
    switch (type)
    {
        case list_by_name:
            qsort(pp, n, sizeof(Person), (pif2vp)cmpname);
        break;

        case list_by_birthday:
            qsort(pp, n, sizeof(Person), (pif2vp)cmpdate);
        break;
    }
    // list the sorted Person-database
}
```

There are several reasons why this setup is not particularly desirable:

- Although the example only shows two list-alternatives (sort by name and sort by birthday), a real-life implementation will have many more ways to list the information. This will soon result in a very long function `list()` which will be hard to maintain and will look inaccessible due to its length.
- Every time a new way to list the data in the database, the function `list()` will have to be expanded, by offering an extra `case` label for every new way to list the data.
- Much of the code in the function `list()` will be repeated within the function, showing only some small differences.

[1] In the current implementation `pp` points to an array of `Person` objects. In this implementation, the function `qsort()` will have to copy the actual `Person` objects again and again, which may be rather inefficient when the `Person` objects become large. Under an alternative implementation, in which the `Person` objects are reached through pointers, the efficiency of the `qsort()` function will be improved. In that case, the datamember `pp` will have to be declared as `Person **pp`.

Much of the complexity of `list()` function could be reduced by defining *pointers* to the compare-functions, storing these pointers in an array. Since this array will be common to all `Person_dbase` objects, it should be defined as a static array, containing the pointers to the compare-functions.

Before actually constructing this array, note that this approach requires the definition of as many compare functions as there are elements in the `Listtype` enum. So, to list the information sorted by name a function `cmpname()` is used, comparing the names stored in two `Person` objects, while a function `cmpcity()`, is used to compare cities. Somehow this seems to be redundant as well: we would like to use one function to compare strings, whatever their meanings. Comparable considerations hold true for other fields of information.

The compare functions, however, receive pointers to `Person` objects. Therefore, the data-members of the `Person` objects to which these pointers point can be accessed using the access-member functions of the `Person` class. So, the compare functions can access these data-members as well, using the pointers to the `Person` objects.

Now note that the access member functions that are used within a particular compare function can be *hard-coded*, by plainly mentioning the accessors to be used, and they can be selected indirectly, by using *pointers* to the accessors to be used.

This latter solution allows us to *merge* compare functions that use the same implementations, but use different accessors: By setting a pointer to the appropriate accessor function just before the compare function is called, one single compare function can be used to compare many different kinds of data stored inside `Person` objects.

The compare functions themselves are used within the context of the `Person_dbase` class, where they are passed to the `qsort()` function. The `qsort()` function, however, is a global function. Consequently, the compare functions can't be ordinary member functions of the class `Person_dbase`, but they must be static members of that class, so they can be passed to the `qsort()` function.

Summarizing what we've got so far, we see that the problem has been broken down as follows:

- The `switch` construction in the `list()` function should be replaced by a call to a function using a pointer to a function.
- The actual function to be used is determined by the value of the selector, which is given to `list()` when it's called.
- The `compare()` functions may be further abstracted by combining those comparing the same types.
- When `compare()` functions are combined, the access member function of the `Person` objects to be used will also be found via an array containing pointers to the access member functions of `Person` objects.
- The `compare()` functions are part of the `Person_dbase` class, but it must also be possible to give their addresses as arguments to `qsort()`. Hence, these functions must be defined as static functions of the class `Person_dbase`.

From this analysis the essential characteristics of the proposed implementation emerge.

For every type of listing, as produced by the function `list()`, the following is required:

- The access member function of the `Person` class to be used.

- The `compare()` function to be used. The `compare()` functions will be *static functions* of the class `Person_dbase`, so that they can be passed over to `qsort()`

This information does not depend on a particular `Person_dbase` object, but is common to all of these objects. Hence it will be stored compile-time in a `static Person_dbase` kind of array.

How will the `compare()` functions know which element of this array to use? The requested index is passed to the `list()` member function as a `Listtype` value. The `list()` function can then save this information in a `static Person_dbase::Listtype` variable for the `compare()` functions to use.

We've analyzed enough. Let's build it this way.

### 10.4.1 Pointers to members: an implementation

- First, the necessary class interfaces are defined. The existence of a class `Date` is assumed, containing overloaded operators like < and > to compare dates. To start with, we present the interface of the class `Person`, omitting all the standard stuff like overloaded assignment operator, (copy) constructors, etc.:

```
#include <stdlib.h>     // for qsort()

class Date;

class Person()
{
    public:
        unsigned length() const;
        unsigned weight() const;
        char const *name() const;
        char const *city() const;
        Date const &birthdate() const;

    private:
        // all necessary data members
};
```

- Next, the class `Person_dbase`. Within this class a `struct CmpPerson` is defined, containing two fields:
  - A pointer to a union of compare functions.
    As the compare functions are static functions of the class `Person_dbase`, pointers to these functions are indiscernible from pointers to functions at the global (`::`) level. The compare functions return `ints` (for `qsort()`), and expect two pointers to `Person const` objects. The field `persons` expects the two pointers to `Person const` objects. The field `voids` is the alternate interpretation, to be used with `qsort()`, instead of the typecast `(pif2vp)`.
  - A field `pf` (pointer to access function) of the nested `union Person_accessor`.
    The types of as many different access functions of the `Person` class as are used in the class are declared in this union.

Access functions returning `ints`, `char const *s` and `Date &s` will be needed. Consequently, the `Person_accessor` union contains these (three) types.

From this `CmpPerson` struct a static array `cmpPerson[]` is constructed. It is a `static Person_dbase` array, making it possible for the compare functions to inspect its elements[2].

Also note the `static Listtype selector`. This variable will be used later in the compare functions to find the actual `Person` access function to be used. Here, then, is the interface of the class `Person_dbase`:

```
class Person_dbase
{
    public:
        enum Listtype
        {
            list_by_length,
            list_by_weight,
            list_by_name,
            list_by_city,
            list_by_birthday,
        };

        // ... constructors etc.

        void list(Listtype type);
                    // list the information
    private:
        struct CmpPerson
        {
            union Compare_function
            {
                int (*persons)// comparing two Persons
                    (Person const *p1, Person const *p2);
                int (*voids)// for qsort()
                    (void const *p1, void const *p2);
            }
                cmp;

            union Person_accessor
            {
                char const
                    *(Person::*cp)() const;
                int
                    (Person::*i)() const;
                Date const
                    &(Person::*d)() const;
            }
                pf;     // to Person's access functions
        };

        static CmpPerson
```

[2]The number of elements of the `cmpPerson[]` array is not specified in the interface: that number is determined compile-time by the compiler, when the static variable `cmpPerson[]` is initialized.

```
            cmpPerson[];
        static Listtype
            selector;

        static int cmpstr(Person const *p1,
                          Person const *p2);

        static int cmpint(Person const *p1,
                          Person const *p2);

        static int cmpdate(Person const *p1,
                           Person const *p2);

        Person
            *pp;    // pointer to the info
        unsigned
            n;      // number of persons stored.
};
```

Next, we define each of the members of the `Person_dbase` class (as far as necessary).

- The `list()` function now only has to do three things:
  - The `Listtype` parameter is copied to `selector`,
  - The function `qsort()` is called. Note the use of the `cmpPerson` array to determine which compare function to use.
  - The information of the `Person`objects is displayed. This part is left for the reader to implement.

```
void Person_dbase::list(Listtype type)
{
    selector = type;
    qsort(pp, n, sizeof(Person), cmpPerson[type].cmp.voids);
    // list the sorted Person-database (to be implemented)
}
```

- The array `cmpPerson[]` is a static array of `CmpPerson` elements. In this example there are five different ways to sort the data. Consequently, there are five elements in the array `cmpPerson[]`. All these elements can be defined and initialized by the compiler. No run-time execution time is needed for this.

  However, note the form of the declaration: the array is defined in the scope of the `Person_dbase` class. Its elements are `CmpPerson`s, also defined in the scope of the `Person_dbase` class. Hence the double mentioning of `Person_dbase`.

```
Person_dbase::CmpPerson
    Person_dbase::cmpPerson[] =
    {
        {       // compare- and access
```

```
            // function to compare length
        cmpint,
        Person::length,
    },

    {       // same for weight
        cmpint,
        Person::weight,
    },

    {       // same for name
        cmpstr,
        Person::name,
    },

    {       // same for city
        cmpstr,
        Person::city,
    },

    {       // same for Date
        cmpdate,
        Person::birthdate,
    },
};
```

- Now only the compare functions remain to be implemented. Although five accessors can be used, only three compare functions are needed.

  The compare functions, being static functions, have access to the `cmpPerson[]` array and to the `Listtype selector` variable. This information is used by the compare functions to call the relevant access member function of the two `Person` objects, pointed to by the parameters of the compare functions.

  For this, the `pointer to member` operator `->*` is used. The element `cmpPerson[selector]` contains the function pointers to the functions to be used: they are the fields `pf`, variant `cp`, `i` or `d`. These fields return a pointer to a particular access function of a `Person` object.

  Through these pointers the functions can be associated to a particular `Person` object using the pointer to member operator. This results in expressions like:

  `p1->*cmpPerson[selector].pf.cp`

  By this time we have the name (i.e., address) of an access function for a particular `Person` object. To call this function, parentheses are needed, one set of parentheses to protect this expression from desintegrating due to the high priority of the second set of parentheses, which are needed for the actual call of the function. Hence, we get:

  `(p1->*cmpPerson[selector].pf.cp)()`

  Finally, here are the three compare functions:

```
    int Person_dbase::cmpstr(Person const *p1, Person const *p2)
```

```
{
    return
    (
        strcmp
        (
            (p1->*cmpPerson[selector].pf.cp)(),
            (p2->*cmpPerson[selector].pf.cp)()
        )
    );
}

int Person_dbase::cmpint(Person const *p1, Person const *p2)
{
    return
    (
        (p1->*cmpPerson[selector].pf.i)()
        -
        (p2->*cmpPerson[selector].pf.i)()
    );
}

int Person_dbase::cmpdate(Person const *p1, Person const *p2)
{
    return
    (
        (p1->*cmpPerson[selector].pf.d)()
        <
        (p2->*cmpPerson[selector].pf.d)() ?
            -1
        :
            (p1->*cmpPerson[selector].pf.d)()
            >
            (p2->*cmpPerson[selector].pf.d)()
    );
}
```

# Chapter 11

# The Standard Template Library, generic algorithms

The `Standard Template Library` (STL) consists of containers, generic algorithms, iterators, function objects, allocators and adaptors. The STL is a general purpose library consisting of algorithms and data structures. The data structures that are used in the algorithms are *abstract* in the sense that the algorithms can be used on (practically) every data type.

The algorithms can work on these abstract data types due to the fact that they are *template* based algorithms. In this chapter the *construction* of these templates in not further discussed (see chapter 17 for that). Rather, the *use* of these template algorithms is the focus of this chapter.

Several parts of the standard template library have already been discussed in the **C++** Annotations. In chapter 8 the abstract containers were discussed, and in section 7.8 function objects and adaptors were covered. Also, *iterators* were mentioned at several places in this document.

The remaining components of the STL will be covered in this chapter. Iterators, and the 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.

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 11.2 of this chapter.

## 11.1 Iterators

terators are an abstraction of pointers. In general, the following holds true of iterators:

- Given an iterator `iter`, `*iter` represents the object the iterator points to (alternatively, `iter->` can be used to reach the object the iterator points to).
- `++iter` or `iter++` advances the iterator to the next element. The notion of advancing an iterator to the next element is consequently applied: several containers have a *reversed* iterator type, in which the `iter++` operation actually reaches an previous element in a sequence.

- For the containers that have their elements stored consecutively in memory *pointer arithmetic* is available as well. This counts out the `list`, but includes the `vector, queue, deque, set` and `map`. For these containers `iter + 2` points to the second element beyond the one to which `iter` points.

The STL containers produce iterators (i.e., type `iterator`) using member functions `begin()` and `end()` and, in the case of reversed iterators (type `reverse_iterator`), `rbegin()` and `rend()`. Standard practice requires the iterator range to be *left inclusive*: the notation `[left, right)` indicates that `left` is an iterator pointing to the first element that is to be considered, while `right` is an iterator pointing just *beyond* the last element to be used. The iterator-range is said to be *empty* when `left == right`.

The following example shows a situation where all elements of a vector of strings are written to `cout` using the iterator range `[begin(), end())`, and the iterator range `[rbegin(), rend())`. Note that the `for-loops` for both ranges are identical:

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

int main(int argc, char **argv)
{
    vector<string>
        args(argv, argv + argc);

    for
    (
        vector<string>::iterator iter = args.begin();
            iter != args.end();
                ++iter
    )
        cout << *iter << " ";

    cout << endl;

    for
    (
        vector<string>::reverse_iterator iter = args.rbegin();
            iter != args.rend();
                ++iter
    )
        cout << *iter << " ";

    cout << endl;

    return (0);
}
```

Furthermore, the STL defines *const_iterator* types to be able to visit a range of the elements in a constant container. Whereas the elements of the vector in the previous example could have been altered, the elements of the vector in the next example are immutable, and `const_iterators` are required:

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

int main(int argc, char **argv)
{
    const vector<string>
        args(argv, argv + argc);

    for
    (
        vector<string>::const_iterator iter = args.begin();
            iter != args.end();
                ++iter
    )
        cout << *iter << " ";

    cout << endl;

    for
    (
        vector<string>::const_reverse_iterator iter = args.rbegin();
            iter != args.rend();
                ++iter
    )
        cout << *iter << " ";

    cout << endl;

    return (0);
}
```

The examples also illustrate the use of plain pointers for iterators. The initialization `vector<string> sarg(argv, argv + argc)` provides the `sarg` vector with a pair of pointer-based iterators: `argv` points to the first element to initialize `sarg` with, `argv + argc` points just beyond the last element to be used, `argv++` reaches the next string. This is a general characteristic of pointers, which is why they too can be used in situations where `iterators` are expected.

The STL defines five types of iterators. These types recur in the generic algorithms, and in order to be able to create a particular type of iterator yourself it is important to know their characteristic. In general, it must be possible to

- test iterators for equality (==)
- test iterators for inequality (!=)
- increment iterators using the prefix or postfix increment operator (++)
- access the element iterators refer to using the dereference operator (*).

**InputIterators:** InputIterators can read elements from a container. The dereference operator is guaranteed to work as an `rvalue` in an expression, not as an `lvalue`. 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, terms like `InputIterator1` and `InputIterator2` may be observed. In these cases the iterator-types (like `InputIterator`) contain numbers to indicate which iterator parameters belong together. E.g., the generic function `inner_product()` has the following prototype:

```
Type inner_product(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, Type init);
```

Here `InputIterator1 first1` and `InputIterator1 last1` are a set of input iterators defining one range, while `InputIterator2 first2` defines the beginning of a second range. Notations like this may be observed with other iterator types as well.

**OutputIterators:** OutputIterators can be used to write to a container. The dereference operator is guaranteed to work as an `lvalue` in an expression, not as an `rvalue`. Instead of an OutputIterator it is also possible to (see below) use a Forward-, Bidirectional- or RandomAccessIterator.

**ForwardIterators:** ForwardIterators combine InputIterators and OutputIterators. They can be used to traverse the container in one direction, for reading and/or writing. Instead of a ForwardIterator it is also possible to (see below) use a Bidirectional- or RandomAccessIterator.

**BidirectionalIterators:** BidirectionalIterators allow the traversal of a container in both directions, for reading and writing. Instead of a BidirectionalIterator it is also possible to (see below) use a RandomAccessIterator. For example, to traverse a list or a deque a BidirectionalIterator may be useful.

**RandomAccessIterators:** RandomAccessIterators provide access to any element of the container at any moment. An algorithm such as `sort()` requires a RandomAccessIterator, and can therefore not be used with lists or maps, which only provide BidirectionalIterators.

The example given with the RandomAccessIterator provides an approach towards iterators: look for the iterator that's required by the (generic) algorithm, and then see whether the datastructure supports the required iterator or not. If not, the algorithm cannot be used with the particular datastructure.

### 11.1.1 Insert iterators

The generic algorithms often require a target container into which the results of the algorithm are deposited. For example, the `copy()` algorithm has three parameters, the first two of them define the range of elements which are visited, and the third parameter defines the first position where the result of the copy operation is to be stored. With the `copy()` algorithm the number of elements that are copied are normally available beforehand, since the number is normally equal to the number of elements in the range defined by the first two parameters, but this does not always hold true. Sometimes the number of resulting elements is different from the number of elements in the initial range. The generic algorithm `unique_copy()` is a case in point: the number of elements which are copied to the destination container is normally not known beforehand.

In situations like these, the `inserter()` adaptor functions may be used to create elements in the destination container when they are needed.

There are three `inserter()` adaptors:

- `back_inserter()` calls the container's `push_back()` insert member to add new elements at the end of the container. E.g.,

  ```
          copy(source.rbegin(), source.rend(), back_inserter(destination));
  ```

  will copy all elements of `source` in reversed order to the back of `destination`.

- `front_inserter()` calls the container's `push_front()` insert member to add new elements at the beginning of the container. E.g.,

  ```
          copy(source.begin(), source.end(), front_inserter(destination));
  ```

  will copy all elements of `source` to the front of the destination container (thereby also reversing the order of the elements).

- `inserter()` calls the container's `insert()` member to add new elements starting at a specified starting point within the container. E.g.,

  ```
          copy(source.begin(), source.end(), inserter(destination,
  destination.begin()));
  ```

  will copy all elements of `source` to the destination container, starting at the beginning of `destination`.

### 11.1.2 istream iterators

The `istream_iterator<Type>()` can be used to define an iterator (pair) for an `istream` object or for a subtype of an `istream`. The general form of the `istream_iterator<Type>()` iterator is:

```
istream_iterator<Type> identifier(istream &inStream)
```

Here, `Type` is the type of the data elements that are to be read from the `istream` stream. `Type` may be any of the types for which the `operator>>()` is defined with `istream` objects.

The default (empty) constructor defines the end of the iterator pair, corresponding to `end-of-stream`. For example,

```
istream_iterator<string> endOfStream;
```

Note that the actual *stream* object which is specified for the begin-iterator is *not* mentioned here.

Using a `back_inserter()` and a set of `istream_iterator<>()`s all strings could be read from `cin` as follows:

```
#include <algorithm>
#include <iterator>
#include <string>
#include <vector>

int main()
{
    vector<string>
        vs;

    copy(istream_iterator<string>(cin), istream_iterator<string>(),
         back_inserter(vs));

    for
    (
        vector<string>::iterator from = vs.begin();
            from != vs.end();
                ++from
    )
        cout << *from << " ";
    cout << endl;

    return (0);
}
```

In the above example, note the use of the anonymous versions of the `istream_iterator`s. Especially note the use of the anonymous default constructor. Instead of using `istream_iterator<string>()` the (non-anonymous) construction

```
    istream_iterator<string>
            eos;

    copy(istream_iterator<string>(cin), eos, back_inserter(vs));
```

could have been used.

The `istream_iterator` iterators is available when the `iterator` header file is included. This is, e.g., the case when `iostream` is included.

### 11.1.3 ostream iterators

The `ostream_iterator<Type>()` can be used to define a destination iterator for an `ostream` object or for a subtype of an `ostream`. 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 *delimiter
```

`Type` is the type of the data elements that are to be written to the `ostream` stream. `Type` may be any of the types for which the `operator<<()` is defined with `ostream` objects. The latter form of the `ostream_iterators` separates the individual `Type` data elements by `delimiter` strings. The former form does not use any delimiters.

The following example shows the use of a `istream_iterators` and an `ostream_iterator` to copy information of a file to another file. A subtlety is the statement `in.unsetf(ios::skipws)`: it resets the `ios::skipws` flag. The consequence of this is that the default behavior of the `operator>>()`, to skip whitespace, is modified. White space characters are simply returned by the operator, and the file is copied unrestrictedly. Here is the program:

```
#include <algorithm>
#include <fstream>
#include <iomanip>

int main(int argc, char **argv)
{
    ifstream
        in(argv[1]);

    in.unsetf(ios::skipws);

    ofstream
        out(argv[2]);

    copy(istream_iterator<char>(in), istream_iterator<char>(),
         ostream_iterator<char>(out));

    return (0);
}
```

The `ostream_iterator` iterators are available when the `iterator` header file is included. This is, e.g., the case when `iostream` is included.

## 11.2 The 'auto_ptr' class

One of the problems using pointers is that strict bookkeeping is required about the memory the pointers point to. When a pointer variable goes out of scope, the memory pointed to by the pointer is suddenly inaccessible, and the program suffers from a memory leak. For example, in the following code, a memory leak is introduced in which 200 `int` values remain allocated:

```
    #include <iostream>
```

```
int main()
{
    for (int idx = 0; idx < 200; ++idx)
    {
        int
            c,
            *ip;

        cin >> c;                  // read an int
        ip = new int(c);           // ip points to int initialized to 'c'
    }                              // no delete-operation
    next();                        // whatever comes next
    return (0);
}
```

The standard way to prevent memory leakage is strict bookkeeping: the programmer has to make sure that the memory pointed to by a pointer is deleted just before the pointer variable dies. In the above example the repair would be:

```
#include <iostream>

int main()
{
    for (int idx = 0; idx < 200; ++idx)
    {
        int
            c,
            *ip;

        cin >> c;                  // read an int
        ip = new int(c);           // ip points to int initialized to 'c'
        delete ip;                 // and delete the allocated memory again
    }
    next();                        // whatever comes next
    return (0);
}
```

When a pointer variable is used to point to *a single value or object*, the bookkeeping becomes less of a burden when the pointer variable is defined as a `auto_ptr` object. The template class `auto_ptr` is available when the header file `memory` is included.

Normally, an `auto_ptr` object is initialized to point to a dynamically created value or object. When the `auto_ptr` object goes out of scope, the memory pointed to by the object is automatically deleted, taking over the programmer's responsibility to delete memory.

Alternative forms to create `auto_ptr` objects are available as well, as discussed in the coming sections.

Note that

- the auto_ptr object cannot be used to point to arrays of objects.
- an auto_ptr object should only point to memory that was made available dynamically, as only dynamically allocated memory can be deleted.
- multiple auto_ptr objects should not be allowed to point to the same block of dynamically allocated memory. Once one auto_ptr object goes out of scope, it deletes the memory it points to, immediately rendering the other objects wild. Ways to prevent this situation are discussed below.

The class auto_ptr has several member functions which can be used to access the pointer itself and to have the auto_ptr point to another block of memory. These member functions are discussed in the following sections as well.

*Note:*

> The memory header file which must be included to use the auto_ptr objects used to be incomplete. A modified memory header file which can be used to replace an incomplete file can be found at ftp://ftp.icce.rug.nl/pub/frank/windows/memory. This file can replace the memory file on **Linux** systems (found in /usr/include/g++), and on computers running **MS-Windows** (found in /usr/include/g++-3 on the current RedHat distribution of Cygnus), if problems are encountered when using the distributed memory file.

### 11.2.1 Defining auto_ptr variables

There are three ways to define auto_ptr objects. Each definition contains the usual <type> specifier between pointed brackets. Concrete examples are given in the coming sections, but an overview of the various possibilities is presented here:

- The basic form initializes an auto_ptr object to a block of memory that's allocated by the new operator:

  ```
  auto_ptr<type> identifier (new-expression);
  ```

  This form is discussed in the next section 11.2.2.
- Another form initializes an auto_ptr object through another auto_ptr object:

  ```
  auto_ptr<type> identifier(another auto_ptr for type);
  ```

  This form is discussed in the next section 11.2.3.
- 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 the next section 11.2.4.

### 11.2.2 Pointing to a newly allocated object

The basic form to initialize an `auto_ptr` object is to pass its constructor a block of memory that's allocated by the `new` operator. The generic form is:

```
auto_ptr<type> identifier (new-expression);
```

For example, to initialize an `auto_ptr` to a `string` variable the construction

```
auto_ptr<string> strPtr (new string("Hello world"));
```

can be used. To initialize an `auto_ptr` to a `double` variable the construction

```
auto_ptr<double> dPtr (new double(123.456));
```

can be used.

Note the use of the operator `new` in the above expressions. The use of the operator `new` ensures the dynamic nature of the memory pointed to by the `auto_ptr` objects, and allows the deletion of the memory once the `auto_ptr` objects go out of scope. Also note that the `type` does *not* contain the pointer: the `type` used in the `auto_ptr` construction is the same type as used in the `new` expression.

In the example of the 200 `int` values given earlier, the memory leak can be avoided by using `auto_ptr` objects as follows:

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

int main()
{
    for (int idx = 0; idx < 200; ++idx)
    {
        int
            c;

        cin >> c;                   // read an int
        auto_ptr<int> ip (new int(c));
    }                               // no delete-operation needed
    return (0);
}
```

Following each cycle of the for loop, the memory allocated by the `new int(c)` expression is deleted automatically.

All member functions that are available for objects that are allocated by the `new` expression (like the `string` object in the first example in this section) can be reached via the `auto_ptr` as if it was a plain pointer to the dynamically allocated object. E.g., to insert some text beyond the wordt `hello` in the string pointed to by `strPtr`, an expression like

```
strPtr->insert(strPtr->find_first_of(" ") + 1, "C++ ");
```

can be used.

### 11.2.3 Pointing to another auto_ptr

Another form to initialize an auto_ptr object is to initialize it from another auto_ptr object for the same type. The generic form is:

```
auto_ptr<type> identifier (other auto_ptr object);
```

For example, to initialize an auto_ptr to a string variable, given the strPtr variable defined in the previous section, the construction

```
auto_ptr<string> newPtr(strPtr);
```

can be used.

A comparable construction can be used with the assignment operator in expressions. One auto_ptr object may be assigned to another auto_ptr object of the same type. For example:

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

int main()
{
    auto_ptr<string>
        hello(new string("Hello world")),
        hello2(hello),
        hello3(new string("Another string"));

    hello3 = hello2;
    return (0);
}
```

Looking at the above example, we see that hello is initialized as described in the previous section. A new expression is used to allocate a string variable dynamically. Next, hello2 is initialized to hello, which is possible, as they are auto_ptr objects of the same types. However, in order to prevent problems when either object goes out of scope, special measures are required.

If the program would stop here, both hello and hello2 go out of scope. But only hello2 would point to the dynamically allocated string hello world: once a auto_ptr object is used to initialize another auto_ptr object, the former (initializing) object does not refer anymore to the allocated string. The string is now 'owned' by the latter (initialized) object.

A comparable action takes place in the assignment statement hello3 = hello2. Here, prior to the actual assignment, the memory pointed to by hello3 is deleted automatically. Then hello3 gains the ownership of the string Hello world, and hello2 cannot be used anymore to reach the string Hello world.

### 11.2.4 Creating an plain auto_ptr

The third form to create an auto_ptr object simply creates an empty auto_ptr object that does not point to a particular block of memory:

```
auto_ptr<type> identifier;
```

In this case the underlying pointer is set to 0 (zero). Since the auto_ptr object itself is not the pointer, its value cannot be compared to 0 to see if it has not been initialized. E.g., code like

```
auto_ptr<int>
    ip;

if (!ip)
    cout << "0-pointer with an auto_ptr object ?" << endl;
```

will not produce any output (actually, it won't compile either...). So, how do we inspect the value of the pointer that's maintained by the auto_ptr object? For this the member get() is available. This member function, as well as the other member functions of the class auto_ptr are described in the following sections.

### 11.2.5 The get() member function

The member function get() of an auto_ptr object returns the underlying pointer. The value returned by get() is a pointer to the underlying data-type. It may be inspected: if it's zero the auto_ptr object does not point to any memory.

The member function get() cannot be used to let the auto_ptr object point to (another) block of memory. Instead the member function reset(), discussed in the next section, should be used.

### 11.2.6 The reset() member function

The member function reset() of an auto_ptr object can be used to (re)assign a block of memory allocated by the operator new to an auto_ptr. The function reset() does not return a value.

An example of its use is:

```
auto_ptr<string>
    str;

str.reset(new string("Hello"));         // assignment of a value
str.reset(new string("Hello world"));   // reassignment of a value
```

The object that is assigned to the pointer using reset() must have been allocated using the new operator. The object the pointer points to just before applying reset()) is deleted first. The value

0 can be passed to `reset()` if the object pointed to by the pointer should be deleted. Following `reset(0)` the pointer variable has been reinitialized.

Note that it is usually more efficient to use a reassignment member function of the object pointed to by the pointer if the only purpose of the exercise is to redefine the value of the object. For example, the `string` class supports a function `assign()` which may be used for that purpose. So, a construction like:

```
        auto_ptr<string>
            aps(new string("Hello"));

        aps.reset("Hello world");
```

can more efficiently be implemented as:

```
        auto_ptr<string>
            aps(new string("Hello"));

        aps->assign("Hello world");
```

### 11.2.7 The release() member function

As we saw in section 11.2.3, when an `auto_ptr` is assigned to another `auto_ptr`, the pointer providing the value loses its value and is reinitialized to 0. If that's not what we want, the member function `release()` may be used.

The `release()` memeberfunction returns the address of the underlying pointer used by the `auto_ptr` object, and releases the ownership of the object at the same time. The ownership can then be taken over by another `auto_ptr` variable (or, indeed, by any other pointer).

In the following example a pointer is initialized, and then another pointer is created to point to the same string as the first `auto_ptr` points to. The first `auto_ptr` still points to the string, but doesn't own the string anymore. Therefore, when the first `auto_ptr` goes out of scope, it won't delete the string pointed to by the second `auto_ptr`.

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

    int main()
    {
        auto_ptr<string>
            first;

        {
            auto_ptr<string>
                second(new string("Hello world"));
```

```
            first.reset(second.release());

            cout << "Second auto_ptr still points at: " << *second << endl
                << "First auto_ptr also points to: " << *first << endl;
        }
        cout << "Second object now out of scope. First auto_ptr\n"
            "still points at: " << *first << endl;
    }
```

## 11.3 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(s)
- The function prototype
- A short description
- A short example.

In the prototypes of the algorithms `Type` is used to specify a generic (i.e., template) datatype. The particular kind of iterator that is required is mentioned, and possibly other generic types, e.g., performing `BinaryOperations`, like `plus<Type>()`.

Almost every generic algorithm has as its first two arguments an iterator range `[first, last)`, defining the range of elements on which the algorithm operates.

### 11.3.1 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: the `operator+()` is applied to all elements implied by the iterator range and to the initial value `init`, and the resulting value is returned.
  - The second prototype: the `op()` is applied to all elements implied by the iterator range and to the initial value `init`, and the resulting value is returned.
- Example:

```
#include<numeric>
#include<vector>
#include<iostream>

int main()
{
    int
        ia[] = {1, 2, 3, 4};
    vector<int>
        iv(ia, ia + 4);

    cout <<
        "Sum of values: " << accumulate(iv.begin(), iv.end(), int(0)) <<
        endl <<
        "Product of values: " << accumulate(iv.begin(), iv.end(), int(1),
                                        multiplies<int>()) <<
        endl;

    return(0);
}
```

### 11.3.2 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:
  - The first prototype: The first returned element is equal to the first element of the input range. The remaining returned elements are equal to the difference of the corresponding element in the input range and its previous element.
  - The second prototype: The first returned element is equal to the first element of the input range. The remaining returned elements are equal to the result of the binary operator `op` applied to the corresponding element in the input range (left operand) and its previous element (right operand).

- Example:

  ```
  #include<numeric>
  #include<vector>
  #include<iostream>

  int main()
  ```

```
{
    int
        ia[] = {1, 2, 5, 10};
    vector<int>
        iv(ia, ia + 4),
        ov(iv.size());

    adjacent_difference(iv.begin(), iv.end(), ov.begin());
    copy(ov.begin(), ov.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    adjacent_difference(iv.begin(), iv.end(), ov.begin(), minus<int>());
    copy(ov.begin(), ov.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return(0);
}
```

### 11.3.3 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 set 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 set of two adjacent elements for which the binary predicate `pred` returns `true` is returned. If no such element exists, `last` is returned.

- Example (see section 11.3.5 for a description of the `copy()` generic algorithm that is used in the following example):

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

  class SquaresDiff
  {
      public:
          SquaresDiff(unsigned minimum): minimum(minimum)
          {}
          bool operator()(unsigned first, unsigned second)
  ```

```
        {
            return (second * second - first * first  >= minimum);
        }
    private:
        unsigned
            minimum;
};


int main()
{
    string
        sarr[] =
        {
            "Alpha", "bravo", "charley", "echo", "echo", "delta",
            "foxtrot", "golf"
        };
    string
        *last = sarr + sizeof(sarr) / sizeof(string),
        *result = adjacent_find(sarr, last);

    cout << *result << endl;
    result = adjacent_find(++result, last);

    cout << "Second time, starting from the next position:\n" <<
        (
            result == last ?
                "** No more adjacent equal elements **"
            :
                "*result"
        ) << endl;

    unsigned
        *ires,
        iv[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
        *ilast = iv + sizeof(iv) / sizeof(unsigned);


    ires = adjacent_find(iv, ilast, SquaresDiff(10));
    cout <<
        "The first numbers for which the squares differ by at least 10 are: "
        << *ires << " and " << *(ires + 1) << endl;

    return(0);
}
```

### 11.3.4 binary_search()

- Header file:

```
#include<algorithm>
```

- Function prototypes:
  - `bool binary_search(ForwardIterator first, ForwardIterator last, Type const &value);`
  - `bool binary_search(ForwardIterator first, ForwardIterator last, Type const &value, Comparator comp);`
- Description:
  - The first prototype: `value` is looked up using binary search in the range of elements implied by the iterator range `[first, last)`. The elements in the range must have been sorted by the `Type::operator<()` function. `True` is returned if the element was found, `false` otherwise.
  - The second prototype: `value` is looked up using binary search in the range of elements implied by the iterator range `[first, last)`. The elements in the range must have been sorted by the `Comparator` function object. `True` is returned if the element was found, `false` otherwise.
- Example:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <functional>

int main()
{
    string
        sarr[] =
        {
            "Alpha", "bravo", "charley", "echo", "delta",
            "foxtrot", "golf", "hotel"
        };
    string
        *last = sarr + sizeof(sarr) / sizeof(string);
    bool
        result = binary_search(sarr, last, "foxtrot");
    cout << (result ? "found " : "didn't find ") << "foxtrot" << endl;

    reverse(sarr, last);                // reverse the order of elements
                                        // binary search now fails:
    result = binary_search(sarr, last, "foxtrot");
    cout << (result ? "found " : "didn't find ") << "foxtrot" << endl;
                                        // ok when using appropriate
                                        // comparator:
    result = binary_search(sarr, last, "foxtrot", greater<string>());
    cout << (result ? "found " : "didn't find ") << "foxtrot" << endl;


    return(0);
}
```

### 11.3.5 copy()

- Header file:

  ```
  #include<algorithm>
  ```

- Function prototype:
  - `OutputIterator copy(InputIterator first, InputIterator last, OutputIterator destination);`

- Description:
  - The range of elements implied by the iterator range `[first, last)` are copied to an output range, starting at `destination`, using the assignment operator of the underlying data type. The returnvalue is the OutputIterator pointing just beyond the last element that was copied to the destinatin range (so, 'last' in the destination range is returned). In the example, note the second call to `copy()`. It uses an `ostream_iterator` for `string` objects. This iterator will write the `string` values to the specified `ostream` (i.e., `cout`), separating the values by the specified separation string (i.e., `" "`).

- Example:

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

int main()
{
    string
        sarr[] =
        {
            "Alpha", "bravo", "charley", "echo", "delta",
            "foxtrot", "golf", "hotel"
        };
    string
        *last = sarr + sizeof(sarr) / sizeof(string);

    copy(sarr + 2, last, sarr);     // move all elements two positions left

                                    // copy to cout using an ostream_iterator
                                    // for strings,
    copy(sarr, last, ostream_iterator<string>(cout, " "));
    cout << endl;

    return(0);
}
```

### 11.3.6 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 returnvalue is the BidirectionalIterator pointing at the last element that was copied to the destinatin range (so, 'first' in the destination range, pointed to by `last2 - (last - first)`, is returned).
- Example:

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

int main()
{
    string
        sarr[] =
        {
            "Alpha", "bravo", "charley", "echo", "delta",
            "foxtrot", "golf", "hotel"
        };
    string
        *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        copy_backward(sarr + 3, last, last - 3),
        last,
        ostream_iterator<string>(cout, " ")
    );
    cout << endl;

    return(0);
}
```

### 11.3.7 count()

- Header file:

```
#include<algorithm>
```

- Function prototypes:
  - `size_t cout(InputIterator first, InputIterator last, Type const &value);`
- Description:
  - The number of times `value` occurs in the iterator range `first, last` is returned. To determine wheter `value` is equal to an element in the iterator range `Type::operator==()` is used.
- Example:

```
#include<algorithm>
#include<iostream>

int main()
{
    int
        ia[] = {1, 2, 3, 4, 3, 4, 2, 1, 3};

    cout << "Number of times the value 3 is available: " <<
        count(ia, ia + sizeof(ia) / sizeof(int), 3) <<
        endl;

    return(0);
}
```

### 11.3.8 count_if()

- Header file:

```
#include<algorithm>
```

- Function prototypes:
  - `size_t cout_if(InputIterator first, InputIterator last, Predicate predicate);`
- Description:
  - The number of times unary predicate `predicate` returns `true` when applied to the elements implied by the iterator range `first, last` is returned.
- Example:

```
#include<algorithm>
#include<iostream>

class Odd
{
    public:
        bool operator()(int value)
        {
            return (value & 1);
```

```
        }
};

int main()
{
    int
        ia[] = {1, 2, 3, 4, 3, 4, 2, 1, 3};

    cout << "The number of odd values in the array is: " <<
        count_if(ia, ia + sizeof(ia) / sizeof(int), Odd()) <<
        endl;

    return(0);
}
```

### 11.3.9 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 corresponsing elements. The ranges need not be of equal length, only the elements in the indicated range are considered (and must be available).
- Example:

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

  class CaseString
  {
      public:
          bool operator()(string const &first, string const &second) const
          {
              return (!strcasecmp(first.c_str(), second.c_str()));
  ```

```
        }
};

int main()
{
    string
        first[] =
        {
            "Alpha", "bravo", "Charley", "echo", "Delta",
            "foxtrot", "Golf", "hotel"
        },
        second[] =
        {
            "alpha", "bravo", "charley", "echo", "delta",
            "foxtrot", "golf", "hotel"
        };
    string
        *last = first + sizeof(first) / sizeof(string);

    cout << "The elements of ‘first’ and ‘second’ are pairwise " <<
        (equal(first, last, second) ? "equal" : "not equal") <<
        endl <<
        "compared case-insensitively, they are " <<
        (equal(first, last, second, CaseString()) ? "equal" : "not equal") <<
        endl;

    return(0);
}
```

### 11.3.10 equal_range()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first, ForwardIterator last, Type const &value);`
  - `pair<ForwardIterator, ForwardIterator> equal_range(ForwardIterator first, ForwardIterator last, Type const &value, Compare comp);`
- Description:
  - The first prototype: Starting from a sorted sequence (where the `operator<()` of the underlying data type was used to sort the elements in the provided range), a pair of iterators representing the returnvalue of, respectively, lower_bound() and upper_bound()is returned.

- The second prototype: Starting from a sorted sequence (where the `comp` function object was used to sort the elements in the provided range), a pair of iterators representing the returnvalue of, respectively, lower_bound() and upper_bound()is returned.

- Example:

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <utility>
#include <vector>

int main()
{
    int
        range[] = {1, 3, 5, 7, 7, 9, 9, 9};
    unsigned const
        size = sizeof(range) / sizeof(int);

    pair<int *, int *>
        pi;

    pi = equal_range(range, range + size, 7);

    cout << "Lower bound for 7: ";
    copy(pi.first, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "Upper bound for 7: ";
    copy(pi.second, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    sort(range, range + size, greater<int>());

    cout << "Sorted in descending order\n";

    copy(range, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    pi = equal_range(range, range + size, 7, greater<int>());

    cout << "Lower bound for 7: ";
    copy(pi.first, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "Upper bound for 7: ";
    copy(pi.second, range + size, ostream_iterator<int>(cout, " "));
    cout << endl;


    return (0);
}
```

### 11.3.11 fill()

- Header file:

```
#include<algorithm>
```

- Function prototypes:
  - `void fill(ForwardIterator first, ForwardIterator last, Type const &value);`
- Description:
  - all the elements implied by the interator range `[first, last)` are initialized to `value`, overwriting the previous values stored in the range.
- Example:

```
#include<algorithm>
#include<vector>
#include<iostream>
#include<iterator>

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);
}
```

### 11.3.12 fill_n()

- Header file:

```
#include<algorithm>
```

- Function prototypes:
  - `void fill_n(ForwardIterator first, Size n, Type const &value);`
- Description:
  - `n` elements starting at the element pointed to by `first` are initialized to `value`, overwriting the previous values stored in the range.
- Example:

```
#include<algorithm>
#include<vector>
#include<iostream>
#include<iterator>

int main()
{
    vector<int>
        iv(8);

    fill_n(iv.begin(), 8, 8);

    copy(iv.begin(), iv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    return(0);
}
```

### 11.3.13 find()

- Header file:

```
#include<algorithm>
```

- Function prototypes:
  - `InputIterator find(InputIterator first, InputIterator last, Type const &value);`
- Description:
  - Element `value` is searched for in the range of the elements implied by the interator range `[first, last)`. An iterator pointing to the first element found is returned. If the element was not found, `last` is returned. The `operator==()` of the underlying data type is used to compare the elements.
- Example:

```
#include<algorithm>
#include<string>
#include<iostream>
#include<iterator>

int main()
{
    string
        sarr[] =
        {
            "alpha", "bravo", "charley", "echo", "delta",
            "foxtrot", "golf", "hotel"
        };
    string
        *last = sarr + sizeof(sarr) / sizeof(string);
```

```
    copy
    (
        find(sarr, last, "echo"), last, ostream_iterator<string>(cout, " ")
    );
    cout << endl;

    if (find(sarr, last, "india") == last)
    {
        cout << "‘india’ was not found in the range\n";
        copy(sarr, last, ostream_iterator<string>(cout, " "));
        cout << endl;
    }

    return(0);
}
```

### 11.3.14 find_if()

- Header file:

```
#include<algorithm>
```

- Function prototypes:
  - `InputIterator find_if(InputIterator first, InputIterator last, Prdicate pred);`
- Description:
  - An iterator pointing to the first element in the range implied by the interator range `[first, last)` for which the (unary) predicate `pred` returns `true` is returned. If the element was not found, `last` is returned.
- Example:

```
#include<algorithm>
#include<string>
#include<iostream>
#include<iterator>

class CaseName
{
    public:
        CaseName(char const *str): _string(str)
        {}
        bool operator()(string const &element)
        {
            return (!strcasecmp(element.c_str(), _string.c_str()));
        }
    private:
        string
            _string;
```

```
};


int main()
{
    string
        sarr[] =
        {
            "Alpha", "Bravo", "Charley", "Echo", "Delta",
            "Foxtrot", "Golf", "Hotel"
        };
    string
        *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        find_if(sarr, last, CaseName("foxtrot")),
        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);
}
```

### 11.3.15 find_end()

- Header file:

  `#include<algorithm>`

- Function prototypes:
  - `ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2)`
  - `ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred)`
- Description:
  - The first prototype: The sequence of elements implied by `[first1, last1)` is searched for the last occurrence of the sequence of elements implied by `[first2, last2)`. If the sequence `[first2, last2)` is not found, `last1` is returned, otherwise an iterator pointing to the first element of the matching sequence is returned. The `operator==()` of the underlying data type is used to compare the elements in the two sequences.

  - The second prototype: The sequence of elements implied by `[first1, last1)` is searched for the last occurrence of the sequence of elements implied by `[first2, last2)`. If the sequence `[first2, last2)` is not found, `last1` is returned, otherwise an iterator pointing to the first element of the matching sequence is returned. The provided binary predicate is used to compare the elements in the two sequences.

- Example:

```
#include<algorithm>
#include<string>
#include<iostream>
#include<iterator>

class Twice
{
    public:
        bool operator()(unsigned first, unsigned second) const
        {
            return (first == (second << 1));
        }
};

int main()
{
    string
        sarr[] =
        {
            "alpha", "bravo", "charley", "echo", "delta",
            "foxtrot", "golf", "hotel",
            "foxtrot", "golf", "hotel",
            "india", "juliet", "kilo"
        },
        search[] =
        {
            "foxtrot",
            "golf",
            "hotel"
        };
    string
        *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (
        find_end(sarr, last, search, search + 3),   // shows sequence starting
        last, ostream_iterator<string>(cout, " ")   // at 2nd 'foxtrot'
    );
    cout << endl;

    unsigned
        range[] = {2, 4, 6, 8, 10, 4, 6, 8, 10},
        nrs[]   = {2, 3, 4};
```

```
    copy                // show sequence of values starting at last sequence
    (                   // of range[] that are twice the values in nrs[]
        find_end(range, range + 9, nrs, nrs + 3, Twice()),
        range + 9, ostream_iterator<unsigned>(cout, " ")
    );
    cout << endl;

    return(0);
}
```

### 11.3.16 find_first_of()

- Header file:

  `#include<algorithm>`

- Function prototypes:

  - `ForwardIterator1 find_first_of(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2)`
  - `ForwardIterator1 find_first_of(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred)`

- Description:

  - The first prototype: The sequence of elements implied by `[first1, last1)` is searched for the first occurrence of an element in the sequence of elements implied by `[first2, last2)`. If no element in the sequence `[first2, last2)` is found, `last1` is returned, otherwise an iterator pointing to the first element in `[first1, last1)` that is equal to an element in `[first2, last2)` is returned. The `operator==()` of the underlying data type is used to compare the elements in the two sequences.
  - The second prototype: The sequence of elements implied by `[first1, first1)` is searched for the first occurrence of an element in the sequence of elements implied by `[first2, last2)`. Each element in the range `[first1, last1)` is compared to each element in the range `[first2, last2)`, and an iterator to the first element in `[first1, last1)` for which the binary predicate `pred` (receiving an the element out of the range `[first1, last1)` and an element from the range `[first2, last2)`) returns `true` is returned. Otherwise, `last1` is returned.

- Example:

```
#include<algorithm>
#include<string>
#include<iostream>
#include<iterator>

class Twice
{
```

```
    public:
        bool operator()(unsigned first, unsigned second) const
        {
            return (first == (second << 1));
        }
};

int main()
{
    string
        sarr[] =
        {
            "alpha", "bravo", "charley", "echo", "delta",
            "foxtrot", "golf", "hotel",
            "foxtrot", "golf", "hotel",
            "india", "juliet", "kilo"
        },
        search[] =
        {
            "foxtrot",
            "golf",
            "hotel"
        };
    string
        *last = sarr + sizeof(sarr) / sizeof(string);

    copy
    (                                               // shows sequence starting
        find_first_of(sarr, last, search, search + 3),  // at 1st 'foxtrot'
        last, ostream_iterator<string>(cout, " ")
    );
    cout << endl;

    unsigned
        range[] = {2, 4, 6, 8, 10, 4, 6, 8, 10},
        nrs[]   = {2, 3, 4};


    copy                // show sequence of values starting at first sequence
    (                   // of range[] that are twice the values in nrs[]
        find_first_of(range, range + 9, nrs, nrs + 3, Twice()),
        range + 9, ostream_iterator<unsigned>(cout, " ")
    );
    cout << endl;

    return(0);
}
```

### 11.3.17 for_each()

- Header file:

```
#include<algorithm>
```

- Function prototype:
  - `Function for_each(InputIterator first, InputIterator last, Function func);`
- Description:
  - Each of the elements implied by the iterator range `[first, last)` is passed in turn to the *function* `func`. The function may not modify the elements it receives (as the used iterator is an input iterator). If the elements are to be transformed, `transform()` (see section 11.3.63) should be used. The function object is returned: see the example below, in which an extra argument list is added to the `for_each()` call, which argument is eventually also passed to the function given to `for_each()`. Within `for_each()` the returnvalue of the function that is passed to it is ignored.
- Example:

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

void capitalizedOutput(string const &str)
{
    char
        *tmp = strcpy(new char[str.size() + 1], str.c_str());

                            // can't use for_each here,
                            // as 'tmp' is modified
    transform(tmp + 1, tmp + str.size(), tmp + 1, tolower);

    tmp[0] = toupper(*tmp);
    cout << tmp << " ";
    delete tmp;
};

int main()
{
    string
        sarr[] =
        {
            "alpha", "BRAVO", "charley", "ECHO", "delta",
            "FOXTROT", "golf", "HOTEL",
        },
        *last = sarr + sizeof(sarr) / sizeof(string);

    for_each(sarr, last, capitalizedOutput)("that's all, folks");
    cout << endl;

    return(0);
```

```
}
```

### 11.3.18 generate()

- Header file:

```
#include<algorithm>
```

- Function prototypes:
  - `void generate(ForwardIterator first, ForwardIterator last, Generator generator);`
- Description:
  - all the elements implied by the interator range `[first, last)` are initialized by the returnvalue of `generator`, which can be a function or function object.
- Example:

```
#include<algorithm>
#include<vector>
#include<iostream>
#include<iterator>

class NaturalSquares
{
    public:
        NaturalSquares(): newsqr(0), last(0)
        {}
        unsigned operator()()
        {                                       // (a + 1)^2 == a^2 + 2*a + 1
            return (newsqr += (last++ << 1) + 1);
        }

    private:
        unsigned
            newsqr,
            last;
};

int main()
{
    vector<unsigned>
        uv(10);

    generate(uv.begin(), uv.end(), NaturalSquares());

    copy(uv.begin(), uv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
```

```
    return(0);
}
```

### 11.3.19 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 interator `first` are initialized by the returnvalue of `generator`, which can be a function or function object.
- Example:

```
#include<algorithm>
#include<vector>
#include<iostream>
#include<iterator>

class NaturalSquares
{
    public:
        NaturalSquares(): newsqr(0), last(0)
        {}
        unsigned operator()()
        {                                       // (a + 1)^2 == a^2 + 2*a + 1
            return (newsqr += (last++ << 1) + 1);
        }

    private:
        unsigned
            newsqr,
            last;
};

int main()
{
    vector<unsigned>
        uv(10);

    generate_n(uv.begin(), 10, NaturalSquares());

    copy(uv.begin(), uv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
```

```
    return(0);
}
```

### 11.3.20 includes()

- Header files:

```
        #include<algorithm>
```

- Function prototypes:
  - `bool includes(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);`
  - `bool includes(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp);`
- Description:
  - The first prototype: Both sequences of elements implied by the ranges `[first1, last1)` and `[first2, last2)` should be sorted, using the `operator<()` of the underlying datatype. The function returns `true` if every element in the second sequence `[first2, second2)` is contained in the first sequence `[first1, second1)` (the second range is a subset of the first range).
  - The second prototype: Both sequences of elements implied by the ranges `[first1, last1)` and `[first2, last2)` should be sorted, using the `comp` function object. The function returns `true` if every element in the second sequence `[first2, second2)` is contained in the first 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()(string const &first, string const &second) const
        {
            return (!strcasecmp(first.c_str(), second.c_str()));
        }
};

int main()
{
    string
        first1[] =
        {
```

```
            "alpha", "bravo", "charley", "echo", "delta",
            "foxtrot", "golf", "hotel"
        },
        first2[] =
        {
            "Alpha", "bravo", "Charley", "echo", "Delta",
            "foxtrot", "Golf", "hotel"
        },
        second[] =
        {
            "charley", "foxtrot", "hotel"
        };
    unsigned
        n = sizeof(first1) / sizeof(string);

    cout << "The elements of `second' are " <<
        (includes(first1, first1 + n, second, second + 3) ? "" : "not")
        << " contained in the first sequence: 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);
}
```

### 11.3.21 inner_product()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `Type inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, Type init);`
  - `Type inner_product(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, Type init, BinaryOperator1 op1, BinaryOperator2 op2);`

- Description:
  - The first prototype: The sum of all pairwise products of the elements implied by the range `[first1, last1)` and the same number of elements starting at the element pointed to by `first2` are added to `init`, and this sum is returned. The function uses the `operator+()` and `operator*()` of the underlying datatype.
  - The second prototype: Binary operator `op2` instead of the default addition operator, and binary operator `op1` instead of the default multiplication operator are applied to all pairwise elements implied by the range `[first1, last1)` and the same number of elements starting at the element pointed to by `first2`. The final result is returned.
- Example:

```
#include <numeric>
#include <algorithm>
#include <iostream>
#include <string>

class Cat
{
    public:
        Cat(string const &sep): sep(sep)
        {}
        string operator()(string const &s1, string const &s2)
        {
            return (s1 + sep + s2);
        }
    private:
        string
            sep;
};

int main()
{
    unsigned
        ia1[] = {1, 2, 3, 4, 5, 6, 7},
        ia2[] = {7, 6, 5, 4, 3, 2, 1},
        init = 0;

    cout << "The sum of all squares in ";
    copy(ia1, ia1 + 7, ostream_iterator<unsigned>(cout, " "));
    cout << "is " <<
        inner_product(ia1, ia1 + 7, ia1, init) << endl;

    cout << "The sum of all cross-products in ";
    copy(ia1, ia1 + 7, ostream_iterator<unsigned>(cout, " "));
    cout << " and ";
    copy(ia2, ia2 + 7, ostream_iterator<unsigned>(cout, " "));
    cout << "is " <<
        inner_product(ia1, ia1 + 7, ia2, init) << endl;

    string
        names1[] = {"Frank", "Karel", "Piet"},
```

```
        names2[] = {"Brokken", "Kubat", "Plomp"};

    cout << "A list of all combined names in ";
    copy(names1, names1 + 3, ostream_iterator<string>(cout, " "));
    cout << "and ";
    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);
}
```

### 11.3.22 inplace_merge()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last);`
  - `void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last, Compare comp);`
- Description:
  - The first prototype: The two (sorted) ranges `[first, middle)` and `[middle, last)` are merged, keeping a sorted list (using the `operator<()` of the underlying data type). The final series is stored in the range `[first, last)`.
  - The second prototype: The two (sorted) ranges `[first, middle)` and `[middle, last)` are merged, keeping a sorted list (using the boolean result of the binaray comparison operator `comp`). The final series is stored in the range `[first, last)`.
- Example:

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

class CaseString
{
    public:
        bool operator()(string const &first, string const &second) const
        {
            return (strcasecmp(first.c_str(), second.c_str()) < 0);
```

```
        }
};

int main()
{
    string
        range[] =
        {
            "alpha", "charley", "delta", "foxtrot", "hotel"
            "bravo", "echo", "golf"
        };

    inplace_merge(range, range + 5, range + 8);
    copy(range, range + 8, 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);
}
```

### 11.3.23 iter_swap()

- Header file:

  `#include<algorithm>`

- Function prototypes:
  - `void iter_swap(ForwardIterator1 iter1, ForwardIterator2 iter2);`

- Description:
  - The elements pointed to by `iter1` and `iter2` are swapped.

- Example:

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

int main()
{
```

```
    string
        first[] = {"alpha", "bravo", "charley", "delta", "echo", "delta"},
        second[] = {"echo", "foxtrot", "golf", "hotel", "india", "kilo"};
    unsigned
        n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    for (unsigned idx = 0; idx < n; ++idx)
        iter_swap(first + idx, second + idx);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.24 lexicographical_compare()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);`
  - `bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, Compare comp);`
- Description:
  - The first prototype: The corresponding pairs of elements in the ranges pointed to by `[first1, last1)` and `[first2, last2)` are compared. The function returns `true`
    * at the first element in the first range which is less than the corresponding element in the second range (using the `operator<()` of the underlying data type),
    * if `last1` is reached, but `last2` isn't reached yet.

    False is returned in the other cases, which indicates that the first sequence is not lexicographical less than the second sequence. I.e., `false` is returned
    * at the first element in the first range which is greater than the corresponding element in the second range (using the `operator<()` of the underlying data type),
    * if `last2` is reached, but `last1` isn't reached yet.

    * if `last1` and `last2` are reached.

  - The second prototype: With this function the binary comparison operation as defined by `comp` is used instead of the underlying `operator<()`.

- Example:

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

class CaseString
{
    public:
        bool operator()(string const &first, string const &second) const
        {
            return (strcasecmp(first.c_str(), second.c_str()) < 0);
        }
};

int main()
{
    char const
        word1[] = "help",
        word2[] = "hello";

    cout << word1 << " is " <<
        (
            lexicographical_compare(word1, word1 + strlen(word1),
                                    word2, word2 + strlen(word2)) ?
                "before "
            :
                "beyond or at "
        ) <<
        word2 << " in the alphabet\n";

    cout << word1 << " is " <<
        (
            lexicographical_compare(word1, word1 + strlen(word1),
                                    word1, word1 + strlen(word1)) ?
                "before "
            :
                "beyond or at "
        ) <<
        word1 << " in the alphabet\n";

    cout << word2 << " is " <<
        (
            lexicographical_compare(word2, word2 + strlen(word2),
                                    word1, word1 + strlen(word1)) ?
                "before "
            :
                "beyond or at "
        ) <<
```

```
        word1 << " in the alphabet\n";

    string
        one[] = {"alpha", "bravo", "charley"},
        two[] = {"ALPHA", "BRAVO", "DELTA"};

    copy(one, one + 3, ostream_iterator<string>(cout, " "));
    cout << " is ordered " <<
        (
            lexicographical_compare(one, one + 3,
                                    two, two + 3, CaseString()) ?
                "before "
            :
                "beyond or at "
        );
    copy(two, two + 3, ostream_iterator<string>(cout, " "));
    cout << endl <<
        "using case-insensitive comparisons.\n";

    return (0);
}
```

### 11.3.25 lower_bound()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const Type &value);`
  - `ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last, const Type &value, Compare comp);`
- Description:
  - The first prototype: The sorted elements implied by the iterator range `[first, last)` are searched for the first element that that is not less than (i.e., greater than or equal to) `value`. The returned iterator marks the location in the sequence where `value` can be inserted without breaking the sorted order of the elements. The `operator<()` of the underlying datatype is used. If no such element is found, `last` is returned.
  - The second prototype: The elements implied by the iterator range `[first, last)` must have been sorted using the `comp` function (-object). Each element in the range is compared to `value` using the `comp` function. An iterator to the first element for which the binary predicate `comp`, applied to the elements of the range and `value`, returns `false` is returned. If no such element is found, `last` is returned.
- Example:

```
#include <algorithm>
#include <iostream>
#include <functional>

int main()
{
    int
        ia[] = {10, 20, 30};

    cout << "Sequence: ";
    copy(ia, ia + 3, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "15 can be inserted before " <<
            *lower_bound(ia, ia + 3, 15) << endl;
    cout << "35 can be inserted after " <<
            (lower_bound(ia, ia + 3, 35) == ia + 3 ?
                                "the last element" : "???") << endl;

    iter_swap(ia, ia + 2);

    cout << "Sequence: ";
    copy(ia, ia + 3, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "15 can be inserted before " <<
            *lower_bound(ia, ia + 3, 15, greater<int>()) << endl;
    cout << "35 can be inserted before " <<
            (lower_bound(ia, ia + 3, 35, greater<int>()) == ia ?
                                "the first element " : "???") << endl;

    return (0);
}
```

### 11.3.26 max()

- Header file:

  `#include<algorithm>`

- Function prototypes:
  - `Type const &max(Type const &one, Type const &two);`
  - `Type const &max(Type const &one, Type const &two, Comparator comp);`
- Description:
  - The first prototype: The larger of the two elements `one` and `two` is returned, using the `operator>()` of the underlying type.
  - The second prototype: `one` is returned if the binary predicate `comp(one, two)` returns `true`, otherwise `two` is returned.
- Example:

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

class CaseString
{
    public:
        bool operator()(string const &first, string const &second) const
        {
            return (strcasecmp(second.c_str(), first.c_str()) > 0);
        }
};


int main()
{
    cout << "Word '" << max(string("first"), string("second")) <<
                                "' is lexicographically last\n";

    cout << "Word '" << max(string("first"), string("SECOND")) <<
                                "' is lexicographically last\n";

    cout << "Word '" << max(string("first"), string("SECOND"),
                        CaseString()) << "' is lexicographically last\n";

    return (0);
}
```

### 11.3.27 max_element()

- Header file:

  ```
  #include<algorithm>
  ```

- Function prototypes:
  - `ForwardIterator max_element(ForwardIterator first, ForwardIterator last);`
  - `ForwardIterator max_element(ForwardIterator first, ForwardIterator last, Comparator comp);`
- Description:
  - The first prototype: An iterator pointing to the largest element in the range implied by `[first, last)` is returned. The `operator>()` of the underlying type is used.
  - The second prototype: rather than using `operator>()`, the binary predicate `comp` is used to make the comparisons between the elements implied by the iterator range `[first, last)`. The element with which `comp` returns most often `true` is returned.
- Example:

  ```
  #include <algorithm>
  #include <iostream>
  ```

```
class AbsValue
{
    public:
        bool operator()(int first, int second) const
        {
            return (abs(second) > abs(first));
        }
};


int main()
{
    int
        ia[] = {-4, 7, -2, 10, -12};

    cout << "The maximum int value is " << *max_element(ia, ia + 5) << endl;
    cout << "The maximum absolute int value is " <<
            *max_element(ia, ia + 5, AbsValue()) << endl;

    return (0);
}
```

### 11.3.28 merge()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `OutputIterator merge(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
  - `OutputIterator merge(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`
- Description:
  - The first prototype: The two (sorted) ranges `[first, middle)` and `[middle, last)` are merged, keeping a sorted list (using the `operator<()` of the underlying data type). The final series is stored in the range starting at `result` and ending just before the OutputIterator that is returned by the function.
  - The second prototype: The two (sorted) ranges `[first, middle)` and `[middle, last)` are merged, keeping a sorted list (using the boolean result of the binaray comparison operator `comp`). The final series is stored in the range starting at `result` and ending just before the OutputIterator that is returned by the function.
- Example:

```
#include <algorithm>
```

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

class CaseString
{
    public:
        bool operator()(string const &first, string const &second) const
        {
            return (strcasecmp(first.c_str(), second.c_str()) < 0);
        }
};

int main()
{
    string
        range1[] =
        {
            "alpha", "bravo", "foxtrot", "hotel", "zulu"
        },
        range2[] =
        {
            "delta", "echo", "golf", "romeo"
        },
        result[5 + 4];


    copy(result,
        merge(range1, range1 + 5, range2, range2 + 4, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string
        range3[] =
        {
            "ALPHA", "bravo", "foxtrot", "HOTEL", "ZULU"
        },
        range4[] =
        {
            "delta", "ECHO", "GOLF", "romeo"
        };


    copy(result,
        merge(range3, range3 + 5, range4, range4 + 4, result, CaseString()),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return(0);
}
```

### 11.3.29 min()

- Header file:

  ```
  #include<algorithm>
  ```

- Function prototypes:
  - `Type const &min(Type const &one, Type const &two);`
  - `Type const &min(Type const &one, Type const &two, Comparator comp);`
- Description:
  - The first prototype: The smaller of the two elements `one` and `two` is returned, using the `operator<()` of the underlying type.
  - The second prototype: `one` is returned if the binary predicate `comp(one, two)` returns `false`, otherwise `two` is returned.
- Example:

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

  class CaseString
  {
      public:
          bool operator()(string const &first, string const &second) const
          {
              return (strcasecmp(second.c_str(), first.c_str()) > 0);
          }
  };


  int main()
  {
      cout << "Word '" << min(string("first"), string("second")) <<
                                  "' is lexicographically first\n";

      cout << "Word '" << min(string("first"), string("SECOND")) <<
                                  "' is lexicographically first\n";

      cout << "Word '" << min(string("first"), string("SECOND"),
                          CaseString()) << "' is lexicographically first\n";

      return (0);
  }
  ```

### 11.3.30 min_element()

- Header file:

  ```
  #include<algorithm>
  ```

- Function prototypes:
  - `ForwardIterator min_element(ForwardIterator first, ForwardIterator last);`
  - `ForwardIterator min_element(ForwardIterator first, ForwardIterator last, Comparator comp);`
- Description:
  - The first prototype: An iterator pointing to the smallest element in the range implied by `[first, last)` is returned. The `operator<()` of the underlying type is used.
  - The second prototype: rather than using `operator<()`, the binary predicate `comp` is used to make the comparisons between the elements implied by the iterator range `[first, last)`. The element with which `comp` returns most often `false` is returned.
- Example:

```
#include <algorithm>
#include <iostream>

class AbsValue
{
    public:
        bool operator()(int first, int second) const
        {
            return (abs(second) > abs(first));
        }
};


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);
}
```

### 11.3.31 mismatch()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);`

- `pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, Compare comp);`

- Description:
  - The first prototype: The two sequences of elements starting at `first1` and `first2` are compared using the equality operator of the underlying data type. Comparison stops if the compared elements differ (i.e., `operator==()` returns false) or `last1` is reached. A `pair` containing iterators pointing to the final positions is returned. The second sequence may contain more elements than the first sequence. The behavior of the algorithm is undefined if the second sequence contains less elements than the first sequence.
  - The second prototype: The two sequences of elements starting at `first1` and `first2` are compared using With this function the binary comparison operation as defined by `comp` is used instead of the underlying `operator==()`. Comparison stops if the `comp` function returns `false` or `last1` is reached. A `pair` containing iterators pointing to the final positions is returned. The second sequence may contain more elements than the first sequence. The behavior of the algorithm is undefined if the second sequence contains less elements than the first sequence.

- Example:

```
#include <algorithm>
#include <string>
#include <iostream>
#include <utility>

class CaseString
{
    public:
        bool operator()(string const &first, string const &second) const
        {
            return (strcasecmp(first.c_str(), second.c_str()) == 0);
        }
};

int main()
{
    string
        range1[] =
        {
            "alpha", "bravo", "foxtrot", "hotel", "zulu"
        },
        range2[] =
        {
            "alpha", "bravo", "foxtrot", "Hotel", "zulu"
        };


    pair<string *, string *>
        pss = mismatch(range1, range1 + 5, range2);

    cout << "The elements " << *pss.first << " and " << *pss.second <<
            " at offset " << (pss.first - range1) << " differ\n";
    if
```

```
    (
        mismatch(range1, range1 + 5, range2, CaseString()).first
        == range1 + 5
    )
        cout << "When compared case-insensitively they match\n";

    return(0);
}
```

### 11.3.32 next_permutation()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);`
  - `bool next_permutation(BidirectionalIterator first, BidirectionalIterator last, Comp comp);`
- Description:
  - The first prototype: The next permutation given the sequence of elements in the range `[first, last)` is determined. For example, if the elements `1, 2` and `3` are the range for which `next_permutation()` is called, then subsequent calls of `next_permutation()` reorders the elements as follows:

    ```
    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 in such a way that each new permutation is obtained by reordering the elements in such a way that the next bigger value is obtained, using the elements < operator. 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, according to the `operator<()` of the underlying data type.
  - The second prototype: The next permutation given the sequence of elements in the range `[first, last)` is determined. The elements in the range are reordered. The value `true` is returned if a reordering took place, the value `false` is returned if no reordering took place, which is the case if the resulting sequence would haven been ordered, using the binary predicate `comp` to compare two elements.
- Example:

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

class CaseString
{
    public:
        bool operator()(string const &first, string const &second) const
        {
            return (strcasecmp(first.c_str(), second.c_str()) < 0);
        }
};

int main()
{
    string
        saints[] = {"Oh", "when", "the", "saints"};

    cout << "All 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);
}
```

### 11.3.33 nth_element()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);`
  - `void nth_element(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last, Compare comp);`
- Description:
  - The first prototype: All elements in the range `[first, last)` are sorted relative to the element pointed to by `nth`: all elements in the range `[left, nth)` are smaller than the element pointed to by `nth`, and 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 underlying datatype is used.
  - The second prototype: All elements in the range `[first, last)` are sorted relative to the element pointed to by `nth`: all elements in the range `[left, nth)` are smaller than the element pointed to by `nth`, and 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 <functional>

int main()
{
    int
        ia[] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};

    nth_element(ia, ia + 3, ia + 10);
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    nth_element(ia, ia + 5, ia + 10, greater<int>());
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.34 partial_sort()

- Header files:

```
#include<algorithm>
```

- Function prototypes:

- void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last);
- void partial_sort(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last, Compare comp);

• Description:

- The first prototype: The `middle - first` smallest elements are sorted and stored in the `[first, middle)`, using the `operator<()` of the underlying datatype. The remaining elements of the series remain unsorted.
- The second prototype: The `middle - first` smallest elements (according to the provided binary predicate `comp` are sorted and stored in the `[first, middle)`. The remaining elements of the series remain unsorted.

• Example:

```
#include <algorithm>
#include <iostream>
#include <functional>

int main()
{
    int
        ia[] = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};

    partial_sort(ia, ia + 3, ia + 10);
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    partial_sort(ia, ia + 5, ia + 10, greater<int>());
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.35 partial_sort_copy()

• Header files:

```
#include<algorithm>
```

• Function prototypes:

- void partial_sort_copy(InputIterator first, InputIterator last, RandomAccessIterator dest_first, RandomAccessIterator dest_last);
- void partial_sort_copy(InputIterator first, InputIterator last, RandomAccessIterator dest_first, RandomAccessIterator dest_last, Compare comp);

• Description:

- The first prototype: The smallest elements in the range `[first, last)` are copied to the range `[dest_first, dest_last)`, using the `operator<()` of the underlying datatype. Only the number of elements in the smaller range are copied to the second range.
- The second prototype: The elements in the range `[first, last)` are are sorted by the binary predicate `comp`. The elements for which the predicate returns most often `true` are copied to the range `[dest_first, dest_last)`. Only the number of elements in the smaller range are copied to the second range.

- Example:

```
#include <algorithm>
#include <iostream>
#include <functional>

int main()
{
    int
        ia[] = {1, 10, 3, 8, 5, 6, 7, 4, 9, 2},
        ia2[6];

    partial_sort_copy(ia, ia + 10, ia2, ia2 + 6);
    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;
    copy(ia2, ia2 + 6, ostream_iterator<int>(cout, " "));
    cout << endl;

    partial_sort_copy(ia, ia + 4, ia2, ia2 + 6);
    copy(ia2, ia2 + 6, ostream_iterator<int>(cout, " "));
    cout << endl;

    partial_sort_copy(ia, ia + 4, ia2, ia2 + 6, greater<int>());
    copy(ia2, ia2 + 6, ostream_iterator<int>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.36 partial_sum()

- Header files:

```
#include<numeric>
```

- Function prototypes:
  - `OutputIterator partial_sum(InputIterator first, InputIterator last, OutputIterator result);`
  - `OutputIterator partial_sum(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation op);`

- Description:
  - The first prototype: the value of each element in the range `[result, <returned OutputIterator>)` is obtained by adding the elements in the corresponding range of the range `[first, last)`. The first element in the resulting range will be equal to the element pointed to by `first`.
  - The second prototype: the value of each element in the range `[result, <returned OutputIterator>)` is obtained by applying the binary operator `op` to the previous element in the resulting range and the corresponding element in the range `[first, last)`. The first element in the resulting range will be equal to the element pointed to by `first`.
- Example:

```
#include <numeric>
#include <algorithm>
#include <iostream>
#include <functional>

int main()
{
    int
        ia[] = {1, 2, 3, 4, 5},
        ia2[5];

    copy(ia2,
        partial_sum(ia, ia + 5, ia2),
        ostream_iterator<int>(cout, " "));
    cout << endl;

    copy(ia2,
        partial_sum(ia, ia + 5, ia2, multiplies<int>()),
        ostream_iterator<int>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.37 partition()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `BidirectionalIterator partition(BidirectionalIterator first, BidirectionalIterator last, UnaryPredicate pred);`
- Description:

  - All elements in the range `[first, last)` for which the unary predicate `pred` evaluates as `true` are placed before the elements which evaluate as `false`. The returnvalue points just beyond the last element in the partitioned range for which `pred` evaluates as `true`.

- Example:

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

class LessThan
{
    public:
        LessThan(int x): x(x)
        {}
        bool operator()(int value)
        {
            return (value <= x);
        }
    private:
        int
            x;
};

int main()
{
    int
        ia[] = {1, 3, 5, 7, 9, 10, 2, 8, 6, 4},
        *split;


    split = partition(ia, ia + 10, LessThan(ia[9]));
    cout << "Last element <= 4 is ia[" << split - ia - 1 << "]\n";

    copy(ia, ia + 10, ostream_iterator<int>(cout, " "));
    cout << endl;
    return (0);
}
```

### 11.3.38 prev_permutation()

- Header files:

```
#include<algorithm>
```

- Function prototypes:

- bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last);
- bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last, Comp comp);

- Description:
  - The first prototype: The previous permutation given the sequence of elements in the range [first, last) is determined. The elements in the range are reordered in such a way that the first ordering is obtained representing a 'smaller' value (see next_permutation() (section 11.3.32 for an example involving the opposite ordering). The value true is returned if a reordering took place, the value false is returned if no reordering took place, which is the case if the provided sequence was already ordered, according to the operator<() of the underlying data type.
  - The second prototype: The previous permutation given the sequence of elements in the range [first, last) is determined. The elements in the range are reordered. The value true is returned if a reordering took place, the value false is returned if no reordering took place, which is the case if the original sequence was already ordered, using the binary predicate comp to compare two elements.

- Example:

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

class CaseString
{
    public:
        bool operator()(string const &first, string const &second) const
        {
            return (strcasecmp(first.c_str(), second.c_str()) < 0);
        }
};

int main()
{
    string
        saints[] = {"Oh", "when", "the", "saints"};

    cout << "All previous permutations of 'Oh when the saints':\n";

    cout << "Sequences:\n";
    do
    {
        copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
        cout << endl;
    }
    while (prev_permutation(saints, saints + 4, CaseString()));


    cout << "After first sorting the sequence:\n";

    sort(saints, saints + 4, CaseString());
```

```
    cout << "Sequences:\n";
    while (prev_permutation(saints, saints + 4, CaseString()))
    {
        copy(saints, saints + 4, ostream_iterator<string>(cout, " "));
        cout << endl;
    }

    cout << "No (more) previous permutations\n";

    return (0);
}
```

### 11.3.39 random_shuffle()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);`
  - `void random_shuffle(RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator rand);`
- Description:
  - The first prototype: The elements in the range `[first, last)` are randomly reordered.
  - The second prototype: The elements in the range `[first, last)` are randomly reordered, using the `rand` random number generator, which should return an `int` in the range `[0, remaining)`, where `remaining` is passed as argument to the `operator()()` of the `rand` function object.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <time.h>

class randomGenerator
{
    public:
        randomGenerator()
        {
            srand(static_cast<int>(time(0)));
```

```
        }
        int operator()(int remaining) const
        {
            return (rand() % remaining);
        }
};

int main()
{
    string
        words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa", "quebec" };
    unsigned
        size = sizeof(words) / sizeof(string);

    random_shuffle(words, words + size);

    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    cout << "sorting the words again\n";
    sort(words, words + size);

    randomGenerator
        rg;
    random_shuffle(words, words + size, rg);

    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.40 remove()

- Header file:

  ```
  #include<algorithm>
  ```

- Function prototype:
  - `ForwardIterator remove(ForwardIterator first, ForwardIterator last, Type &value);`
- Description:
  - The elements in the range pointed to by `[first, last)` are reordered in such a way that all values unequal to `value` are placed at the beginning of the range. The returned forward iterator points to the first element, after reordering, that can be removed. The range `[returnvalue, last)` is called the *leftover* of the algorithm. The leftover may contain other values than `value`, but these can also safely be removed, as they are also present in the range `[first, returnvalue)`.
- Example:

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

int main()
{
    string
        words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" },
        *removed;
    unsigned
        size = sizeof(words) / sizeof(string);

    cout << "Removing all \"alpha\"s:\n";
    removed = remove(words, words + size, "alpha");
    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
         << "Trailing elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.41 remove_copy()

- Header file:

  ```
  #include<algorithm>
  ```

- Function prototypes:
  - `OutputIterator remove_copy(InputIterator first, InputIterator last, OutputIterator result, Type &value);`
- Description:
  - The elements in the range pointed to by `[first, last)` not matching `value` are copied to the range `[result, returnvalue)`, where `returnvalue` is the value returned by the function. The range `[first, last)` is not modified.
- Example:

  ```
  #include <algorithm>
  #include <iostream>
  #include <string>
  #include <vector>
  #include <functional>

  class EqualAlpha
  {
      public:
  ```

```
        operator()(string const &word) const
        {
            return (word == "alpha");
        }
};

int main()
{
    string
        words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    unsigned
        size = sizeof(words) / sizeof(string);
    string
        remaining[size - count_if(words, words + size, EqualAlpha())],
        *returnvalue;

    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);
}
```

### 11.3.42 remove_if()

- Header file:

```
#include<algorithm>
```

- Function prototypes:
  - `ForwardIterator remove_if(ForwardIterator first, ForwardIterator last, UnaryPredicate pred);`
- Description:
  - The elements in the range pointed to by `[first, last)` are reordered in such a way that all values for which the unary predicate `pred` evaluates as `false` are placed at the beginning of the range. The returned forward iterator points to the first element, after reordering, for which `pred` returns `true`. The range `[returnvalue, last)` is called the *leftover* of the algorithm. The leftover may contain other values than `value`, but these can also safely be removed, as they are also present in the range `[first, returnvalue)`.
- Example:

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

```
class Remover
{
    public:
        bool operator()(string const &str)
        {
            return (str == "alpha");
        }
};

int main()
{
    string
        words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" },
        *removed;
    unsigned
        size = sizeof(words) / sizeof(string);

    cout << "Removing all \"alpha\"s:\n";
    removed = remove_if(words, words + size, Remover());
    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);
}
```

### 11.3.43 remove_copy_if()

- Header file:

```
#include<algorithm>
```

- Function prototypes:
  - `OutputIterator remove_copy_if(InputIterator first, InputIterator last, OutputIterator result, UnaryPredicate pred);`
- Description:
  - The elements in the range pointed to by `[first, last)` for which the unary predicate `pred` returns `true` are copied to the range `[result, returnvalue)`, where `returnvalue` is the value returned by the function. The range `[first, last)` is not modified.
- Example:

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

```
#include <vector>
#include <functional>

class EqualAlpha
{
    public:
        bool operator()(string const &word) const
        {
            return (word == "alpha");
        }
};

int main()
{
    string
        words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    unsigned
        size = sizeof(words) / sizeof(string);
    string
        remaining[size - count_if(words, words + size, EqualAlpha())],
        *returnvalue;

    returnvalue = remove_copy_if(words, words + size, remaining, EqualAlpha());

    cout << "Removing all \"alpha\"s:\n";
    copy(remaining, returnvalue, ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.44 replace()

- Header file:

```
#include<algorithm>
```

- Function prototypes:
  - `ForwardIterator replace(ForwardIterator first, ForwardIterator last, Type &oldvalue, Type &newvalue);`
- Description:
  - All elements equal to `oldvalue` in the range pointed to by `[first, last)` are replaced by the value `newvalue`.
- Example:

```
#include <algorithm>
#include <iostream>
```

```
#include <string>

int main()
{
    string
        words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" },
        *removed;
    unsigned
        size = sizeof(words) / sizeof(string);

    replace(words, words + size, string("alpha"), string("ALPHA"));
    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.45 replace_copy()

- Header file:

  ```
  #include<algorithm>
  ```

- Function prototypes:

  - `OutputIterator replace_copy(InputIterator first, InputIterator last, OutputIterator result, Type &oldvalue, Type &newvalue);`

- Description:

  - All elements equal to `oldvalue` in the range pointed to by `[first, last)` are replaced by the value `newvalue` in a new range `[result, returnvalue)`, where `returnvalue` is the returnvalue of the function.

- Example:

  ```
  #include <algorithm>
  #include <iostream>
  #include <string>
  #include <vector>
  #include <functional>
  int main()
  {
      string
          words[] =
          { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
              "oscar", "alpha", "alpha", "papa", "quebec" };
      unsigned
          size = sizeof(words) / sizeof(string);
      string
          remaining[size],
  ```

```
        *returnvalue;

    returnvalue = replace_copy(words, words + size, remaining,
                            string("alpha"), string("ALPHA"));

    copy(remaining, returnvalue, ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.46 replace_if()

- Header file:

  ```
  #include<algorithm>
  ```

- Function prototypes:

  - `ForwardIterator replace_if(ForwardIterator first, ForwardIterator last, UnaryPredicate pred, Type const &value);`

- Description:

  - The elements in the range pointed to by `[first, last)` for which the unary predicate `pred` evaluates as `true` are replaced by `newvalue`.

  Example:

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

class Replacer
{
    public:
        bool operator()(string const &str)
        {
            return (str == "alpha");
        }
};

int main()
{
    string
        words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    unsigned
        size = sizeof(words) / sizeof(string);

    replace_if(words, words + size, Replacer(), string("ALPHA"));
    copy(words, words + size, ostream_iterator<string>(cout, " "));
```

```
    cout << endl;

    return (0);
}
```

### 11.3.47 replace_copy_if()

- Header file:

  ```
  #include<algorithm>
  ```

- Function prototypes:
  - `OutputIterator replace_copy_if(ForwardIterator first, ForwardIterator last, OutputIterator result, UnaryPredicate pred, Type const &value);`

- Description:
  - The elements in the range pointed to by `[first, last)` are copied to the range `[result, returnvalue)`, where `returnvalue` is the value returned by the function. The elements for which the unary predicate `pred` returns `true` are replaced by `newvalue`. The range `[first, last)` is not modified.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <functional>

class Replacer
{
    public:
        bool operator()(string const &str) const
        {
            return (str == "alpha");
        }
};


int main()
{
    string
        words[] =
        { "kilo", "alpha", "lima", "mike", "alpha", "november", "alpha",
            "oscar", "alpha", "alpha", "papa", "quebec" };
    unsigned
        size = sizeof(words) / sizeof(string);
    string
        result[size];

    replace_copy_if(words, words + size, result, Replacer(), string("ALPHA"));
```

```
    copy (result, result + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.48 reverse()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `void reverse(BidirectionalIterator first, BidirectionalIterator last);`
- Description:
  - The elements in the range pointed to by `[first, last)` are reversed.
- Example:

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

int main()
{
    string
        line;

    while (getline(cin, line))
    {
        reverse(line.begin(), line.end());
        cout << line << endl;
    }

    return (0);
}
```

### 11.3.49 reverse_copy()

- Header files:

```
#include<algorithm>
```

- Function prototypes:

– `OutputIterator reverse_copy(BidirectionalIterator first, BidirectionalIterator last, OutputIterator result);`

- Description:
  - The elements in the range pointed to by `[first, last)` are copied to the range `[result, returnvalue)` in reversed order. The value `returnvalue` is the value that is returned by the function.

- Example:

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

int main()
{
    string
        line;

    while (getline(cin, line))
    {
        unsigned
            size = line.size();
        char
            copy[size + 1];

        cout << "line: " << line << endl <<
                "reversed: ";
        reverse_copy(line.begin(), line.end(), copy);
        copy[size] = 0;     // 0 is not part of the reversed
                            // line !
        cout << copy << endl;
    }

    return (0);
}
```

### 11.3.50 rotate()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `void rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);`

- Description:

- The elements implied by the range `[first, middle)` are moved to the end of the container, the elements implied by the range `[middle, last)` are moved to the beginning of the container.

- Example:

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

int main()
{
    string
        words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa", "quebec",
          "echo", "foxtrot", "golf", "hotel", "india", "juliet" };
    unsigned const
        size = sizeof(words) / sizeof(string),
        midsize = 7;

    rotate(words, words + midsize, words + size);

    copy(words, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.51 rotate_copy()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last, OutputIterator result);`

- Description:
  - The elements implied by the range `[middle, last)` and then the elements implied by the range `[first, middle)` are copied to the destination container having range `[result, returnvalue)`, where `returnvalue` is the iterator returned by the function.

- Example:

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

```
int main()
{
    string
        words[] =
        { "kilo", "lima", "mike", "november", "oscar", "papa", "quebec",
          "echo", "foxtrot", "golf", "hotel", "india", "juliet" };
    unsigned const
        size = sizeof(words) / sizeof(string),
        midsize = 7;
    string
        out[size];



    copy(out,
        rotate_copy(words, words + midsize, words + size, out),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.52 search()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);`
  - `ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);`
- Description:
  - The first prototype: An iterator into the first range `[first1, last1)` is returned where the elements in the range `[first2, last2)` are found, using the `operator==()` operator of the underlying data type. If no such location exists, `last1` is returned.
  - The second prototype: An iterator into the first range `[first1, last1)` is returned where the elements in the range `[first2, last2)` are found, using the provided binary predicate `pred` to compare the elements in the two ranges. If no such location exists, `last1` is returned.
- Example:

```
#include <algorithm>
#include <iostream>
```

```
class absInt
{
    public:
        bool operator()(int i1, int i2)
        {
            return (abs(i1) == abs(i2));
        }
};

int main()
{
    int
        range1[] =
            {-2, -4, -6, -8, 2, 4, 6, 8},
        range2[] =
            {6, 8};

    copy
    (
        search(range1, range1 + 8, range2, range2 + 2),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;

    copy
    (
        search(range1, range1 + 8, range2, range2 + 2, absInt()),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;


    return (0);
}
```

### 11.3.53 search_n()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `ForwardIterator1 search_n(ForwardIterator1 first1, ForwardIterator1 last1, Size count, Type const & value);`
  - `ForwardIterator1 search_n(ForwardIterator1 first1, ForwardIterator1 last1, Size count, Type const & value, BinaryPredicate pred);`

- Description:
  - The first prototype: An iterator into the first range `[first1, last1)` is returned where `n` elements having value `value` are found, using the `operator==()` operator of the underlying data type to compare the elements. If no such location exists, `last1` is returned.
  - The second prototype: An iterator into the first range `[first1, last1)` is returned where `n` elements having value `value` are found, using the provided binary predicate `pred` to compare the elements. If no such location exists, `last1` is returned.
- Example:

```
#include <algorithm>
#include <iostream>

class absInt
{
    public:
        bool operator()(int i1, int i2)
        {
            return (abs(i1) == abs(i2));
        }
};

int main()
{
    int
        range1[] =
            {-2, -4, -4, -6, -8, 2, 4, 4, 6, 8},
        range2[] =
            {6, 8};

    copy
    (
        search_n(range1, range1 + 8, 2, 4),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;

    copy
    (
        search_n(range1, range1 + 8, 2, 4, absInt()),
        range1 + 8,
        ostream_iterator<int>(cout, " ")
    );
    cout << endl;


    return (0);
}
```

### 11.3.54 set_difference()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `OutputIterator set_difference( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
  - `OutputIterator set_difference( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`
- Description:
  - The first prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are not present in the range `[first2, last2)` is returned, starting at `[result)`, and ending at the outputiterator that is returned by the function. The elements in the two ranges must have been sorted using the `operator<()` of the underlying datatype.
  - The second prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are not present in the range `[first2, last2)` is returned, starting at `[result)`, and ending at the outputiterator that is returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.
- Example:

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

class CaseLess
{
    public:
        bool operator()(string const &left, string const &right)
        {
            return (strcasecmp(left.c_str(), right.c_str()) < 0);
        }
};

int main()
{
    string
        set1[] =
        { "kilo", "lima", "mike", "november",
            "oscar", "papa", "quebec" },
        set2[] =
        { "papa", "quebec", "romeo"},
        result[7],
        *returned;
```

```
    copy(result,
        set_difference(set1, set1 + 7, set2, set2 + 3, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string
        set3[] =
        { "PAPA", "QUEBEC", "ROMEO"};
    copy(result,
        set_difference(set1, set1 + 7, set3, set3 + 3, result,
        CaseLess()),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.55 set_intersection()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `OutputIterator set_intersection( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
  - `OutputIterator set_intersection( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`
- Description:
  - The first prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are also present in the ranges `[first2, last2)` is returned, starting at `[result)`, and ending at the outputiterator that is returned by the function. The elements in the two ranges must have been sorted using the `operator<()` of the underlying datatype.
  - The second prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are also present in the ranges `[first2, last2)` is returned, starting at `[result)`, and ending at the outputiterator that is returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.
- Example:

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

```
class CaseLess
{
    public:
        bool operator()(string const &left, string const &right)
        {
            return (strcasecmp(left.c_str(), right.c_str()) < 0);
        }
};

int main()
{
    string
        set1[] =
        { "kilo", "lima", "mike", "november",
            "oscar", "papa", "quebec" },
        set2[] =
        { "papa", "quebec", "romeo"},
        result[7],
        *returned;

    copy(result,
        set_intersection(set1, set1 + 7, set2, set2 + 3, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string
        set3[] =
        { "PAPA", "QUEBEC", "ROMEO"};
    copy(result,
        set_intersection(set1, set1 + 7, set3, set3 + 3, result,
        CaseLess()),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.56 set_symmetric_difference()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `OutputIterator set_symmetric_difference( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
  - `OutputIterator set_symmetric_difference( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`

- Description:
  - The first prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are not present in the range `[first2, last2)` and those in the range `[first2, last2)` that are not present in the range `[first1, last1)` is returned, starting at `[result)` and ending at the outputiterator that is returned by the function. The elements in the two ranges must have been sorted using the `operator<()` of the underlying datatype.
  - The second prototype: a sorted sequence of the elements a sorted sequence of the elements pointed to by the range `[first1, last1)` that are not present in the range `[first2, last2)` and those in the range `[first2, last2)` that are not present in the range `[first1, last1)` is returned, starting at `[result)` and ending at the outputiterator that is returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.
- Example:

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

class CaseLess
{
    public:
        bool operator()(string const &left, string const &right)
        {
            return (strcasecmp(left.c_str(), right.c_str()) < 0);
        }
};

int main()
{
    string
        set1[] =
        { "kilo", "lima", "mike", "november",
            "oscar", "papa", "quebec" },
        set2[] =
        { "papa", "quebec", "romeo"},
        result[7],
        *returned;

    copy(result,
        set_symmetric_difference(set1, set1 + 7, set2, set2 + 3, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string
        set3[] =
        { "PAPA", "QUEBEC", "ROMEO"};
    copy(result,
        set_symmetric_difference(set1, set1 + 7, set3, set3 + 3, result,
        CaseLess()),
        ostream_iterator<string>(cout, " "));
```

```
    cout << endl;

    return (0);
}
```

### 11.3.57 set_union()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `OutputIterator set_intersection( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`
  - `OutputIterator set_intersection( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`
- Description:
  - The first prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are also present in the ranges `[first2, last2)` is returned, starting at `[result)`, and ending at the outputiterator that is returned by the function. The elements in the two ranges must have been sorted using the `operator<()` of the underlying datatype.
  - The second prototype: a sorted sequence of the elements pointed to by the range `[first1, last1)` that are also present in the ranges `[first2, last2)` is returned, starting at `[result)`, and ending at the outputiterator that is returned by the function. The elements in the two ranges must have been sorted using the `comp` function object.
- Example:

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

class CaseLess
{
    public:
        bool operator()(string const &left, string const &right)
        {
            return (strcasecmp(left.c_str(), right.c_str()) < 0);
        }
};

int main()
{
    string
        set1[] =
```

```
        { "kilo", "lima", "mike", "november",
            "oscar", "papa", "quebec" },
        set2[] =
        { "papa", "quebec", "romeo"},
        result[7],
        *returned;

    copy(result,
        set_intersection(set1, set1 + 7, set2, set2 + 3, result),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    string
        set3[] =
        { "PAPA", "QUEBEC", "ROMEO"};
    copy(result,
        set_intersection(set1, set1 + 7, set3, set3 + 3, result,
        CaseLess()),
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.58 sort()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `void sort( RandomAccessIterator first, RandomAccessIterator last);`
  - `void sort( RandomAccessIterator first, RandomAccessIterator last, Compare comp);`
- Description:
  - The first prototype: the elements in the range `[first, last)` are sorted in ascending order, using the `operator<()` of the underlying datatype.
  - The second prototype: the elements in the range `[first, last)` are sorted in ascending order, using the `comp` function object to compare the elements.
- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
```

```
int main()
{
    string
        words[] =
        {"november", "kilo", "mike", "lima",
            "oscar", "quebec", "papa"};

    sort(words, words + 7);
    copy(words, words + 7,
        ostream_iterator<string>(cout, " "));
    cout << endl;

    sort(words, words + 7, greater<string>());
    copy(words, words + 7,
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.59 stable_partition()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `BidirectionalIterator stable_partition(BidirectionalIterator first, BidirectionalIterator last, UnaryPredicate pred);`
- Description:
  - All elements in the range `[first, last)` for which the unary predicate `pred` evaluates as `true` are placed before the elements which evaluate as `false`. The relative order of the elements in the container is kept. The returnvalue points just beyond the last element in the partitioned range for which `pred` evaluates as `true`.
- Example:

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

class LessThan
{
    public:
        LessThan(int x): x(x)
        {}
        bool operator()(int value)
        {
```

```
            return (value <= x);
        }
    private:
        int
            x;
};

int main()
{
    int
        org[] = {1, 3, 5, 7, 9, 10, 2, 8, 6, 4},
        ia[10],
        *split;


    copy(org, org + 10, ia);
    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;

    copy(org, org + 10, ia);
    split = stable_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);
}
```

### 11.3.60 stable_sort()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `void stable_sort( RandomAccessIterator first, RandomAccessIterator last);`
  - `void stable_sort( RandomAccessIterator first, RandomAccessIterator last, Compare comp);`
- Description:

- The first prototype: the elements in the range `[first, last)` are stable_sorted in ascending order, using the `operator<()` of the underlying datatype. The relative order of the equal elements is kept.
- The second prototype: the elements in the range `[first, last)` are stable_sorted in ascending order, using the `comp` function object to compare the elements. The relative order of the equal elements is kept.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>

class CmpFirst
{
    public:
        bool operator()(string const &left, string const &right)
        {
            return (left[0] < right[0]);
        }
};

int main()
{
    string
        words[] =
        {"piper", "november", "kilo", "mooney", "mike", "lima",
            "oscar", "quebec", "papa", "netherlands"};

    stable_sort(words, words + 10, CmpFirst());
    copy(words, words + 10,
        ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.61 swap()

- Header file:

```
#include<algorithm>
```

- Function prototypes:
  - `void swap(Type &object1, Type &object2);`
- Description:
  - The elements `object1` and `object2` change values.

- Example:

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

int main()
{
    string
        first[] = {"alpha", "bravo", "charley", "delta", "echo", "delta"},
        second[] = {"echo", "foxtrot", "golf", "hotel", "india", "kilo"};
    unsigned
        n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    for (unsigned idx = 0; idx < n; ++idx)
        swap(first[idx], second[idx]);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.62 swap_ranges()

- Header file:

```
#include<algorithm>
```

- Function prototypes:
  - `ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 result);`
- Description:
  - The elements in the ranges pointed to by `[first1, last1)` are swapped with the elements in the ranges `[result, returnvalue)`, where `returnvalue` is the value returned by the function. The two ranges must be disjoint.
- Example:

```
#include <algorithm>
#include <iostream>
```

```
#include <string>

int main()
{
    string
        first[] = {"alpha", "bravo", "charley", "delta", "echo", "delta"},
        second[] = {"echo", "foxtrot", "golf", "hotel", "india", "kilo"};
    unsigned
        n = sizeof(first) / sizeof(string);

    cout << "Before:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    swap_ranges(first, first + n, second);

    cout << "After:\n";
    copy(first, first + n, ostream_iterator<string>(cout, " "));
    cout << endl;
    copy(second, second + n, ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.63 transform()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `OutputIterator transform(InputIterator first, InputIterator last, OutputIterator result, UnaryOperator op);`
  - `OutputIterator transform(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, OutputIterator result, BinaryOperator op);`
- Description:
  - The first prototype: the unary operator `op` is applied to each of the elements in the range `[first, last)`, and the resulting values are stored in the range starting at `result`. The returnvalue points just beyond the last generated element.
  - The second prototype: the binary operator `op` is applied to each of the elements in the range `[first, last)` and the corresponding element in the second range starting at `first2`. The resulting values are stored in the range starting at `result`. The returnvalue points just beyond the last generated element.
- Example:

```
#include <functional>
#include <vector>
#include <algorithm>
#include <iostream>
#include <string>
#include <ctype.h>

class Caps
{
    public:
        string operator()(string const &src)
        {
            string
                tmp = src;

            transform(&tmp[0], &tmp[tmp.size()], &tmp[0], toupper);
            return (tmp);
        }
};

int main()
{
    string
        words[] = {"alpha", "bravo", "charley"};

    copy(words, transform(words, words + 3, words, Caps()),
            ostream_iterator<string>(cout, " "));
    cout << endl;

    int
        values[] = {1, 2, 3, 4, 5};
    vector<int>
        squares;

    transform(values, values + 5, values,
                back_inserter(squares), multiplies<int>());

    copy(squares.begin(), squares.end(),
            ostream_iterator<int>(cout, " "));
    cout << endl;
    return (0);
}
```

### 11.3.64 unique()

- Header file:

  ```
  #include<algorithm>
  ```

- Function prototypes:
  - `ForwardIterator unique(ForwardIterator first, ForwardIterator last);`
  - `ForwardIterator unique(ForwardIterator first, ForwardIterator last, BinaryPredicate pred);`
- Description:
  - The first prototype: Consecutively equal elements (according to the `operator==()` of the underlying data type) in the range pointed to by `[first, last)` are collapsed into a single element. The returned forward iterator marks the *leftover* of the algorithm, and contains (unique) elements appearing earlier in the range.
  - The second prototype: Consecutive elements in the range pointed to by `[first, last)` for which the binary predicate `pred` returns `true` are collapsed into a single element. The returned forward iterator marks the *leftover* of the algorithm, and contains (unique) elements appearing earlier in the range.
- Example:

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

class CaseString
{
    public:
        bool operator()(string const &first, string const &second) const
        {
            return (!strcasecmp(first.c_str(), second.c_str()));
        }
};

int main()
{
    string
        words[] =
        {"alpha", "alpha", "Alpha", "papa", "quebec" },
        *removed;
    unsigned
        size = sizeof(words) / sizeof(string);

    removed = unique(words, words + size);
    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
         << "Trailing elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;

    removed = unique(words, words + size, CaseString());
    copy(words, removed, ostream_iterator<string>(cout, " "));
    cout << endl
         << "Trailing elements are:\n";
    copy(removed, words + size, ostream_iterator<string>(cout, " "));
    cout << endl;
```

```
    return (0);
}
```

### 11.3.65 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 (according to the `operator==()` of the underlying data type) are copied only once. The returned output iterator points just beyond the last element that was copied.
  - The second prototype: The elements in the range `[first, last)` are copied to the resulting container, starting at `result`. Consecutive elements in the range pointed to by `[first, last)` for which the binary predicate `pred` returns `true` are copied only once. The returned output iterator points just beyond the last element that was copied.

- Example:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <functional>

class CaseString
{
    public:
        bool operator()(string const &first, string const &second) const
        {
            return (!strcasecmp(first.c_str(), second.c_str()));
        }
};

int main()
{
    string
        words[] = {"oscar", "Alpha", "alpha", "alpha", "papa", "quebec" };
    unsigned
        size = sizeof(words) / sizeof(string);
    vector<string>
```

```
        remaining;

    unique_copy(words, words + size,
                        back_inserter(remaining));

    copy(remaining.begin(), remaining.end(),
            ostream_iterator<string>(cout, " "));
    cout << endl;

    vector<string>
        remaining2;

    unique_copy(words, words + size,
                        back_inserter(remaining2), CaseString());

    copy(remaining2.begin(), remaining2.end(),
            ostream_iterator<string>(cout, " "));
    cout << endl;

    return (0);
}
```

### 11.3.66 upper_bound()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const Type &value);`
  - `ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last, const Type &value, Compare comp);`
- Description:
  - The first prototype: The sorted elements implied by the iterator range `[first, last)` are searched for the first element that that is greater than `value`. The returned iterator marks the location in the sequence where `value` can be inserted without breaking the sorted order of the elements. The `operator<()` of the underlying datatype is used. If no such element is found, `last` is returned.
  - The second prototype: The elements implied by the iterator range `[first, last)` must have been sorted using the `comp` function (-object). Each element in the range is compared to `value` using the `comp` function. An iterator to the first element for which the binary predicate `comp`, applied to the elements of the range and `value`, returns `true` is returned. If no such element is found, `last` is returned.
- Example:

```
#include <algorithm>
#include <iostream>
#include <functional>

int main()
{
    int
        ia[] = {10, 20, 30};

    cout << "Sequence: ";
    copy(ia, ia + 3, ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "15 can be inserted before " <<
            *upper_bound(ia, ia + 3, 15) << endl;
    cout << "35 can be inserted after " <<
            (upper_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 " <<
            *upper_bound(ia, ia + 3, 15, greater<int>()) << endl;
    cout << "35 can be inserted before " <<
            (upper_bound(ia, ia + 3, 35, greater<int>()) == ia ?
                                "the first element " : "???") << endl;

    return (0);
}
```

### 11.3.67 Heap algorithms

A heap is a form of binary tree represented as an array. In the standard heap, the key of an element is greater or equal to the key of its children. This kind of heap is called a *max heap.*

A tree in which numbers are keys could be organized as follows:

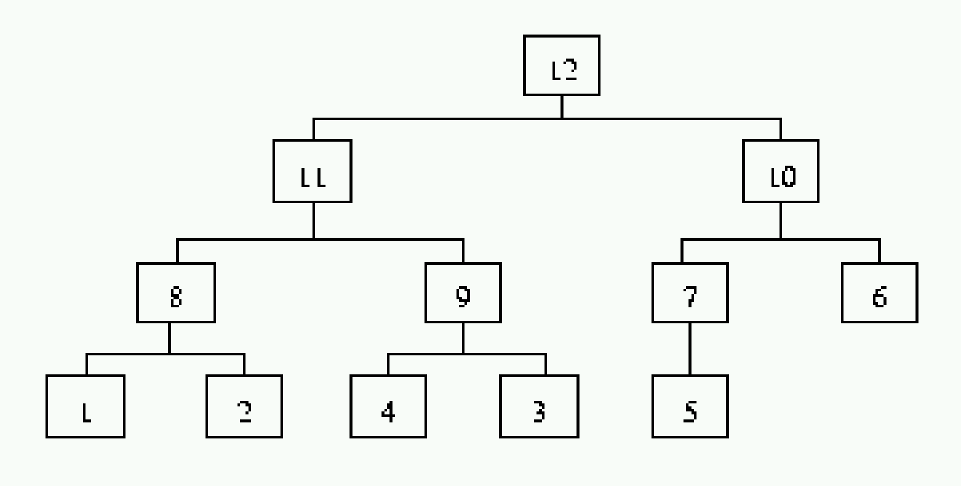
This tree can be organized in an array as follows:

```
12, 11, 10, 8, 9, 7, 6, 1, 2, 4, 3, 5
```

Here, 12 is the top node. its children are 11 and 10, both less than 12. 11, in turn, has 8 and 9 as its children, while the children of 10 are 7 and 6. 8 has 1 and 2 as its children, 9 has 4 and 3, and finally, 7 has left child 5. 7 doesn't have a right child, and 6 has no children.

Note that the left and right branches are not ordered: 8 is less than 9, but 7 is larger than 6.

The heap is formed by traversing a binary tree level-wise, starting from the top node. The top




\

Figure 11.1: A binary tree representation of a heap

node is 12, at the zeroth level. At the first level we find 11 and 10. At the second level 6, 7, 8 and 9 are found, etc.

Heaps can be created in containers supporting random access. So, a heap is not, for example, constructed in a list. Heaps can be constructed from an (unsorted) array (using make_heap()). The top-element can be pruned from a heap, followed by reordering the heap (using pop_heap()), a new element can be added to the heap, followed by reordering the heap (using push_heap()), and the elements in a heap can be sorted (using sort_heap(), which invalidates the heap, though).

The following subsections introduce the prototypes of the heap-algorithms, the final subsection provides a small example in which the heap algorithms are used.

### make_heap()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `void make_heap(RandomAccessIterator first, RandomAccessIterator last);`
  - `void make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`
- Description:
  - The first prototype: The elements in the range `[first, last)` are reordered to form a max-heap, using the `operator<()` of the underlying data type.
  - The second prototype: The elements in the range `[first, last)` are reordered to form a heap, using the binary comparison function object `comp` to compare elements.
- Follow this link for a small example of a program using `make_heap()`.

### pop_heap()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `void pop_heap(RandomAccessIterator first, RandomAccessIterator last);`
  - `void pop_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`
- Description:
  - The first prototype: The first element in the range `[first, last)` is moved to `last - 1`. Then, the elements in the range `[first, last - 1)` are reordered to form a max-heap, using the `operator<()` of the underlying data type.
  - The second prototype: The first element in the range `[first, last)` is moved to `last - 1`. Then, the elements in the range `[first, last - 1)` are reordered to form a heap, using the binary comparison function object `comp` to compare elements.
- Follow this link for a small example of a program using `pop_heap()`.

### push_heap()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `void push_heap(RandomAccessIterator first, RandomAccessIterator last);`
  - `void push_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`
- Description:
  - The first prototype: Assuming that the range `[first, last - 2)` contains a valid heap, and the element at `last - 1` contains an element to be added to the heap, the elements in the range `[first, last - 1)` are reordered to form a max-heap, using the `operator<()` of the underlying data type.
  - The second prototype: Assuming that the range `[first, last - 2)` contains a valid heap, and the element at `last - 1` contains an element to be added to the heap, the elements in the range `[first, last - 1)` are reordered to form a heap, using the binary comparison function object `comp` to compare elements.
- Follow this link for a small example of a program using `push_heap()`.

### sort_heap()

- Header files:

```
#include<algorithm>
```

- Function prototypes:
  - `void sort_heap(RandomAccessIterator first, RandomAccessIterator last);`
  - `void sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);`
- Description:
  - The first prototype: Assuming the elements in the range `[first, last)` form a valid max-heap, the elements in the range `[first, last)` are sorted, using the `operator<()` of the underlying data type.
  - The second prototype: Assuming the elements in the range `[first, last)` form a valid heap, the elements in the range `[first, last)` are sorted, using the binary comparison function object `comp` to compare elements.
- Follow this link for a small example of a program using `sort_heap()`.

### A small example using the heap algorithms

```
#include <algorithm>
#include <iostream>
#include <functional>

void show(int *ia, char const *header)
{
    cout << header << ":\n";
    copy(ia, ia + 20, ostream_iterator<int>(cout, " "));
    cout << endl;
}

int main()
{
    int
        ia[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                11, 12, 13, 14, 15, 16, 17, 18, 19, 20};

    make_heap(ia, ia + 20);
    show(ia, "The values 1-20 in a max-heap");

    pop_heap(ia, ia + 20);
    show(ia, "Removing the first element (now at the end)");

    push_heap(ia, ia + 20);
    show(ia, "Adding 20 (at the end) to the heap again");
```

```
    sort_heap(ia, ia + 20);
    show(ia, "Sorting the elements in the heap");


    make_heap(ia, ia + 20, greater<int>());
    show(ia, "The values 1-20 in a heap, using > (and beyond too)");

    pop_heap(ia, ia + 20, greater<int>());
    show(ia, "Removing the first element (now at the end)");

    push_heap(ia, ia + 20, greater<int>());
    show(ia, "Adding 20 (at the end) to the heap again");

    sort_heap(ia, ia + 20, greater<int>());
    show(ia, "Sorting the elements in the heap");

    return (0);
}
```

# Chapter 12

# The IO-stream Library

As an extension to the standard stream (`FILE`) approach well known from the **C** programming language, **C++** offers an *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*. In this chapter we'll cover the library to a larger extent.

Apart from defining the *insertion (<<)* and *extraction(>>)* operators, the use of the **C++** *I/O library* offers the additional advantage of *type safety* in all kinds of standard situations. Objects (or plain values) are inserted into the *iostreams*. 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 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, the *iostreams* offer more or less the same set of possibilities as the standard streams of **C**: files can be opened, closed, positioned, read, written, etc.. The remainder of this chapter presents an overview.

In general, input is managed by `istream` objects, having the derived classes `ifstream` for files, and `istrstream` for strings (character arrays), whereas output is managed by `ostream` objects, having the derived classes `ofstream` for files and `ostrstream` for strings.

If a file should allow both reading from and writing to, a `fstream` object (c.q. `strstream` object) should be used.

Finally, in order to use the `iostream` facilities, the header file `iostream` must be included in source files using these facilities. When `ifstream, ofstream` or `fstream` objects are to be used, the `fstream` header file, which in turn includes `iostream`, must be included. An analogous situation holds true for string streams. Here the header file `strstream` is required.

## 12.1 Streams: insertion (<<) and extraction (>>)

The insertion and extraction operators are used to write information to or read information from, respectively, *ostream* and *istream* objects (and to all classes derived from these classes). By default, *white space* is skipped when the insertion and extraction operators are used.

### 12.1.1 The insertion operator <<

The *insertion* operator (<<) points to the `ostream` object wherein the information is inserted. The *extraction* operator points to the object receiving the information obtained from the `istream` object.

As an example, the << operator as defined with the `class ostream` is an overloaded operator having as prototype, e.g.,

```
ostream &ostream::operator <<(char const *text)
```

The normal associativity of the <<-operator remains unaltered, so when a statement like

(cout << "hello " << "world")

is encountered, the leftmost two operands are evaluated first (`cout << "hello "`), and a `ostream &` object, which is actually the same `cout` object. From here, the statement is reduced to

(cout << "world")

and the second string is inserted into `cout`.

Since the << operator has a lot of (overloaded) variants, many types of variables can be inserted into `ostream` objects. There is an overloaded <<-operator expecting an `int`, a `double`, a pointer, etc. etc.. For every part of the information that is inserted into the stream the operator returns the `ostream` object into which the information so far was inserted, and the next part of the information to be inserted is devoured.

As we have seen in the discussion of *friends*, even new classes can contain an overloaded << operator to be used with `ostream` objects (see sections 14.3 and 14.3.1).

### 12.1.2 The extraction operator >>

With the *extraction* operator, a similar situation holds true as with the insertion operator, the extraction operator operating comparably to the `scanf()` function. I.e., white space characters are skipped. Also, the operator doesn't expect *pointers* to variables that should be given new values, but *references* (with the exception of the `char *`, but `string` variables are used as references).

Consider the following code:

```
int
    i1,
    i2;
char
    c;

cin >> i1 >> i2;                // see (1)

while (cin >> c && c != '.')    // see (2)
    process(c);

char                            // see (3)
    buffer[80];
                                // see (3)
while (cin >> buffer)
    process(buffer);
```

This example shows several characteristics of the extraction operator worth noting. Assume the input consists of the following lines:

```
125
22
h e l l o
w o r l d .
this example shows
that we're not yet done
with C++
```

1. In the first part of the example two int values are extracted from the input: these values are assigned, respectively, to `i1` and `i2`. White-space (newlines, spaces, tabs) is skipped, and the values 125 and 22 are assigned to `i1` and `i2`.

   If the assignment *fails*, e.g., when there are no numbers to be converted, the result of the extraction operator evaluates to a zero result, which can be used for testing purposes, as in:

   ```
   if (!(cin >> i1))
   ```

2. In the second part, characters are read. However, white space is skipped, so the characters of the words `hello` and `world` are produced by `cin`, but the blanks that appear in between are not.

   Furthermore, the final `'.'` is not processed, since that one's used as a sentinel: the delimiter to end the `while`-loop, when the extraction is still successful.

3. In the third part, the argument of the extraction operator is yet another type of variable: when a `char *` is passed, white-space delimited strings are extracted. So, here the words `this, example, shows, that, we're, not, yet, done, with` and `C++` are returned.

   Then, the end of the information is reached. This has two consequences: First, the `while`-loop terminates. Second, an empty string is copied into the `buffer` variable.

## 12.2 Four standard iostreams

In **C** three standard files are available: `stdin`, the standard input stream, normally connected to the keyboard, `stdout`, the (buffered) standard output stream, normally connected to the screen, and `stderr`, the (unbuffered) standard error stream, normally not redirected, and also connected to the screen.

In **C++** comparable iostreams are

- `cin`, an `istream` object from which information can be extracted. This stream is normally connected to the keyboard.
- `cout`, an `ostream` object, into which information can be inserted. This stream is normally connected to the screen.
- `cerr`, an `ostream` object, into which information can be inserted. This stream is normally connected to the screen. Insertions into that stream are unbuffered.
- `clog`, an `ostream` object, comparable to `cerr`, but using buffered insertions. Again, this stream is normally connected to the screen.

## 12.3 Files and Strings in general

In order to be able to create `fstream` objects, the header file `fstream` must be included. Files to read are accessed through `ifstream` objects, files to write are accessed through `ofstream` objects. Files may be accessed for reading and writing as well. The general `fstream` object is used for that purpose.

String stream objects can be used to read or write objects to streams in memory, allowing the use of, e.g., the insertion and extraction operators on these objects. To use the string stream objects `istrstream, ostrstream` or `strstream` the header file `strstream` must be included. Note that a `strstream` object is not a `string` object. A `strstream` object should be approached like a `fstream` object, not as a `char *` object having special characteristics.

### 12.3.1 String stream objects: a summary

Strings can be processed similarly to `iostream` objects, if objects of the class `istrstream, ostrstream` or `strstream` are constructed. Objects of these classes can be used to, respectively, read information from memory, write information to memory, or both.

These objects can be created by constructors expecting the address of a block of memory (and its size) as its argument. It is also possible to let the objects to the memory management themselves.

Let's go through some examples. To write something into a block of memory using a `ostrstream` object, the following code could be used:

```
char
    buffer[100];
ostrstream
```

```
    os(buffer, 100);    // construct the ostrstream object

                        // fill 'buffer' with a well-known text
os << "Hello world " << endl << ends;

cout << os.str(); // display the string
```

Note the final `ends` that is appended: When an ascii-z string is inserted into an `ostrstream` object it will not automatically write a trailing ascii-z sentinel (comparable to the way `ostream` objects behave). In order to append a terminating ascii-z, the symbolic value `ends` can be used. After inserting an `ends` further insertions into the `ostrstream` object will succeed, but they will not normally be visible:

```
char
    buffer[100];
ostrstream
    os(buffer, 100);    // construct the ostrstream object

os << "Hello world " << ends;

os << " More text " << ends;

cout << os.str() << endl; // this only shows 'Hello world'
```

The information, however, *is* stored in the string, as shown by the following example:

```
void bytes(ostrstream &str)
{
    char
        *cp = str.str();

    cout << str.pcount() << ": ";

    for (int idx = 0; idx < 10; ++idx)
        cout << setw(3) << static_cast<int>(cp[idx]) << " ";
    cout << endl;
}

int main()
{
    char buffer[10];

    memset(buffer, 10, 10);

    ostrstream
        str(buffer, 100);

    bytes(str);
```

```
        str << "A";

        bytes(str);

        str << "B" << ends;

        bytes(str);

        str << "C";

        bytes(str);

        return (0);
    }
```

This little program produces the following output:

```
0:  10  10  10  10  10  10  10  10  10  10
1:  65  10  10  10  10  10  10  10  10  10
3:  65  66   0  10  10  10  10  10  10  10
4:  65  66   0  67  10  10  10  10  10  10
```

This output shows that all insertions succeed, but the `ends` writes an ascii-z character. This effectively creating an ascii-z string, preventing the display of the information beyond when the contents of the `ostrstream` object are inserted into `cout`.

Furthermore, note the use of the member function `str()`, returning the string the `ostrstream` object operates on. Using `str()` the existence of `buffer` can be hidden from the users of the `ostrstream` object.

When an `ostrstream` object is created without an external memory buffer (e.g., '`ostrstream str;`' is defined), the `ostrstream` object allocates the required memory itself. In that case using the `str()` member function will result in the *freezing* of the `ostrstream` object: it will no longer create room for new characters when additional text is inserted into the object, and, most important, it will *not delete allocated memory* when the object itself is deleted.

To prevent memory leakage here, the program using the `str()` member function can take two actions:

- First, as `str()` returns a `char *` rather than a `char const *` the caller of `str()` may consider the returned string its own. Consequently, the caller of `str()` is responsible for deleting the string returned by `str()`. E.g.,

```
    ostrstream
        ostr;

    ostr << "Hello world" << ends;
```

```
char
    *cp = ostr.gets();  // freezes ostr

cout << cp;             // use ostr's string
delete cp;              // caller deletes ostr's string
```

- Alternatively, the string can be *unfrozen*, after which insertions are again possible. Now, when the `ostrstream` object is destroyed the `ostrstream`'s internally stored string is destroyed too. E.g.,

```
ostrstream
    ostr;

ostr << "Hello world" << ends;

char
    *cp = ostr.gets();  // freezes ostr

cout << cp;             // use ostr's string

ostr.freeze(0);         // ostr will now delete its own string, cp
                        // should leave the memory it points to alone.
```

The following member functions are available for `strstream` objects:

- `istrstream::istrstream(const char *str [, int size])`: This constructor creates an input string class `istrstream` object, associating it with an existing buffer starting at `str`, of size `size`. If `size` is not specified, the buffer is treated as a null-terminated string.
- `ostrstream::ostrstream()`: This constructor creates a new stream for output to a dynamically managed string, which will grow as needed.
- `ostrstream::ostrstream(char *str, int size [, int mode])`: This constructor creates a new stream for output to a statically defined string of length `size`, starting at `str`. The `mode` parameter may optionally be specified as one of the iostream modes. By default `ios::out` is used.
- `int ostrstream::pcount()`: returns the current length of the string associated with this `ostrstream` object.
- `char *ostrstream::str()`: The member function returns a pointer to the string managed by this `ostrstream` object. This function implies `freeze()`, see below:
- `void ostrstream::freeze ([int n])`: If `n` is nonzero (the default), the string associated with this `ostrstream` object must not change dynamically anymore. While frozen, it will not be reallocated if it needs more space, and it will not be deallocated when the `ostrstream` object is destroyed. `freeze(1)` can be used to refer to the string as a pointer after creating it via `ostrstream` facilities. `freeze(0)` can be used to unfreeze (thaw ?) the object again. Following `freeze(0)` the `ostrstream` object will delete memory it allocated when the object itself is deleted.

- `int ostrstream::frozen()`: This member can be used to test whether `freeze(1)` is in effect for this string.

In order to use the `strstream` classes, the header file `strstream` must be included.

### 12.3.2 Writing streams

In order to be able to write to a file an `ofstream` object must be created, in order to be able to write to a string stream an `ostrstream` object must be created.

To open a file to write to, the `ofstream` constructor receives the name of the file to be opened:

```
ofstream out("outfile");
```

By default this will result in the creation of the file, and information inserted into it will be written from the beginning of the file. Actually, this corresponds to the creation of the `ofstream` object in standard output mode, for which the enumeration value `ios::out` could have been provided as well:

```
ofstream out("outfile", ios::out);
```

Alternatively, instead of (re)writing the file, the `ofstream` object could be created in the append mode, using the `ios::app` mode indicator:

```
ofstream out("outfile", ios::app);
```

Normally, information will be inserted into the `ofstream` object using the insertion operator <<, in the way it is used with the standard streams like `cout`, e.g.:

```
out << "Information inserted into the 'out' stream\n";
```

Just like the `fopen()` function of **C** may fail, the construction of the `ofstream` object might not succeed. When an attempt is made to create an `ofstream` object, it is a good idea to test the successful construction. The `ofstream` object returns 0 if its construction failed. This value can be used in tests, and the code can throw an exception (see chapter 13) or it can handle the failure itself, as in the following code:

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

int main()
{
    ofstream
        out("/");   // creating 'out' fails

    if (!out)
```

```
        {
            cerr << "creating ofstream object failed\n";
            exit(1);
        }
    }
```

Alternatively, a `ofstream` object may be constructed first, and opened later:

```
    ofstream
        out;

    out.open("outfile");
```

Here, the return value of `open()` may be inspected to see whether the stream has been successfully opened or not.

Analogous to an `ofstream` object, an `ostrstream` object can be created. Here no filename is required. E.g.,

```
ostrstream text;
```

opens an empty `ostrstream` object. There is no `open()` member function for `ostrstream` objects.

An `ostrstream` object may be initialized by an ascii-z string. E.g.,

```
ostrstream text("hello world");
```

These strings expand dynamically when more information is inserted into them. However, the inserted information is not automatically ascii-z terminated. In order to append an ascii-z to the information inserted into an `ostrstream` object an `ends` can be inserted:

```
text << ", and there is more." << ends;
```

The information that is stored in a `ostrstream` object can be retrieved from its `str()` member, which returns a `char const *`, but realize that this will 'freeze' the object, see section 12.3.1. The number of characters returned by `str()` is obtained from the `pcount()` member, returning an `int`.

### 12.3.3 Reading streams

In order to be able to read from a file an `ifstream` object must be created, in order to be able to read from a string stream an `istrstream` object must be created.

To open a file to read from, the `ifstream` constructor receives the name of the file to be opened:

```
ifstream in("infile");
```

By default this will result in the opening of the file for reading. The file must exist for the `ifstream` object construction to succeed. Instead of the shorthand form to open a file for reading, and explicit `ios` flag may be used as well:

```
ifstream in("infile", ios::in);
```

As with the `ofstream` objects, `ifstream` objects may be constructed first, and opened later:

```
ifstream
    ifstr;

ifstr.open("infile");
```

Normally, information will be extracted from the `ifstream` object using the extraction operator >>, in the way it is used with the standard stream `cin`, e.g.:

```
in >> x >> y;
```

By default, the extraction operator skips blanks: between words, between characters, between numbers, etc.. Consequently, if the input consists of the following information:

```
12
13
a b
hello world
```

then the next code fragment will read `12` and `13` into `x` and `y`, will then return the characters `a` and `b`, and will finally read `hello` and `world` into the character array `buffer`:

```
int
    x,
    y;
char
    c,
    buffer[10];

in >> x >> y >> c >> c >> buffer >> buffer;
```

Notice that no format specifiers are necessary. The type of the variables receiving the extracted information determines the nature of the extraction: integer values for `int`s, white space delimited strings for `char []`s, etc..

Just like the `fopen()` function of **C** may fail, the construction of the `ifstream` object might not succeed. When an attempt is made to create an `ifstream` object, it is a good idea to test the successful construction. The `ifstream` object returns 0 if its construction failed. This value can be used in tests, and the code can throw an exception (see section 13) or it can handle the failure itself, as in the following code:

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

int main()
{
    ifstream
        in("");         // creating 'in' fails

    if (!in)
    {
        cerr << "creating ifstream object failed\n";
        exit(1);
    }
}
```

Analogous to an `ifstream` object, an `istrstream` object can be created. Here no filename is required. E.g.,

```
istrstream text("hello world");
```

opens an `istrstream` object that is initialized by an ascii-z string.

### 12.3.4 Reading and writing streams

In order to be able to read and write to a file a `fstream` object must be created. To read and write to a `strstream` a `strstream` object must be created. Again, the constructor receives the name of the file to be opened:

```
fstream inout("infile", ios::in | ios::out);
```

Note the use of the `ios` constants `ios::in` and `ios::out`, indicating that the file must be opened both for reading and writing. Multiple mode indicators may be used, concatenated by the binary or operator `'|'`. Alternatively, instead of `ios::out`, `ios::app` might have been used, in which case writing will always be done at the end of the file.

Under **DOS**-like operating systems, which use the multiple character `\r\n` sentinels to separate lines in textfiles the flag `ios::binary` (or `ios::bin`) is required for processing binary files to ensure that `\r\n` combinations are processed as two characters.

With `fstream` objects, the `ios::out` will result in the creation of the file, if the file doesn't exist, and if `ios::out` is the only mode specification of the file. If the mode `ios::in` is given as well, then the file is created only if it doesn't exist. So, we have the following possibilities:

```
-------------------------------------------------------------
                             Specified Filemode
                ---------------------------------------------
 File:              ios::out           ios::in | ios::out
-------------------------------------------------------------
 exists          File is rewritten    File is used as found

doesn't exist      File is created        File is created
-------------------------------------------------------------
```

Once a file has been opened in read and write mode, the << operator may be used to write to the file, while the >> operator may be used to read from the file. These operations may be performed in random order. The following fragment will read a blank-delimited word from the file, will write a string to the file, just beyond the point where the string just read terminated, and will read another string: just beyond the location where the string just written ended:

```
...
fstream
    f("filename", ios::in | ios::out);
char
    buffer[80]; // for now assume this
                // is long enough

f >> buffer;    // read the first word

                // write a well known text
f << "hello world";

f >> buffer;    // 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 >> buffer` should produce a `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 >> buffer << "grandpa" >> buffer;
```

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.

The member function `tellg()` returns the current offsetposition of the stream for which it is called.

The member function `seekg()` expects two arguments, the second one having a default value:

```
seekg(long offset, seek_dir position = ios::beg);
```

The first argument is a `long` offset with respect to a `seek_dir` postion. The `seek_dir` position may be one of:

- `ios::beg`: add `offset` to the begin of file position. Negative offsets result in an error condition, which must be cleared before any further operations on the file will succeed.
- `ios::end`: add `offset` to the end of file position. Positive offsets result in the insertion of as many padding `(char)0` characters as necessary to reach the intended offset.
- `ios::cur`: add `offset` to the current file position. If adding the `offset` to the current position would result in a position before `ios::beg`, then, again, an error condition results. If the position would be beyond `ios::end`, then extra `(char)0` characters are supplied.

Error conditions (see also section 12.3.6) 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 continues. E.g.,

```
...
fstream
    f("filename", ios::in | ios::out);
char
    buffer[80]; // for now assume this
                // is long enough

f.seekg(-10);   // this fails, but...
f.clear();      // processing f continues

f >> buffer;    // read the first word
```

`Strstream` objects can be given flags as well. The `ostrstream` object may be constructed by the following constructor:

```
ostrstream text(initext, size, flags);
```

where `initext` is an ascii-z terminated initialization text, `size` is the size of the internal buffer of the `strstream` object, and `flags` is a set of `ios` flags. The last and last two arguments are optional. If `size` is specified, the internal buffer will not grow dynamically, but will be given a static size of `size` bytes.

### 12.3.5 Special functions

Special functions are available for unformated (also called *binary*) reading and writing of files.

For *reading* the following member functions, that can be used with `istream` objects (which includes `ifstream` and `fstream` objects) are available:

- `int istream::get()`: this function returns `EOF` or 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`.
- `int istream::peek()`: this function returns the next available input character, but does not actually remove the character from the input stream.
- `istream& istream::get(char *buffer, int len [, char delimiter])`: This function reads a string 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. The delimiter is a newline (`'\n'`) character by default.
  Note that the **delimiter is not read** by `get()` if the delimiter is encountered in the input stream.
  After reading the string into `buffer`, an `ascii-Z` character is appended to the string in `buffer` delimiter.
  Note that the functions `eof()` and `fail()` (see section 12.3.6) do *not* return `true` if the delimiter was not encountered on input. Furthermore, an `ascii-Z` character can be used for the delimiter: this way strings terminating in `ascii-Z` characters may be read from a (binary) file. Of course, the program reading the string should know in advance the number of characters that are going to be read by the function. A small example illustrating the use of `get()` in combination with `ascii-Z` delimiters is:

```
#include <fstream>

void main(int argc, char **argv)
{
    fstream
        f(argv[1], ios::in | ios::out); // r/w the file, assume this
                                        // succeeds
    f.write("hello", 6);
    f.write("hello", 6);

    f.seekg(0, ios::beg);               // reset to begin of file

    char
        buffer[20],
        c;
                                        // read the first 'hello'
    cout << f.get(buffer, 100, 0).tellg() << endl;;
    f >> c;                             // read the ascii-z delim

                                        // and read the second 'hello'
    cout << f.get(buffer + 6, 100, 0).tellg() << endl;;
}
```

- `istream& istream::getline (char *buffer, int len [, char delimiter])` This function operates analogously to `get()`, but `delimiter` is removed from the input 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 `delimiter` was *not* found (before reading `len` characters or `EOF`, `getline()` the `fail()` member function, and possibly also `eof()` will return true.

- `istream& istream::read(void *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 writing *binary* files. In the following example, an object of the class `Date` is written in its binary form to a file:

```
#include <fstream>

class Date
{
    public:
        unsigned
            year,
            month,
            day;
};

void main(int argc, char **argv)
{
    ofstream
        f(argv[1], ios::out);
    Date
        d;
    f.read(&d, sizeof(Date));  // reads object in binary form.
}
```

- `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 unformated input operation.

- `istream& istream::ignore ([int n [, int delimiter]])`: called without arguments, one character is skipped from the input stream. Called with one argument, `n` characters are skipped. The optional second argument specifies a delimiter: after skipping `n` or the `delimiter` character (whichever comes first) the function returns.

- `istream& istream::putback (char c)`: the character `c` is 'pushed back' into the input stream, to be read as the next character. `EOF` is returned if this is not allowed. One character may always be put back.

- `istream& istream::unget()`: an attempt is made to push back the last character that was read.

For *writing* binary files the following member functions are available. These functions can be used with `ostream` objects (including `ofstream` and `fstream` objects):

- `ostream& ostream::put(char c)`: this function writes a single character to the output stream.
- `ostream& ostream::write (void *buffer, int length)`: this function writes at most `len` bytes to the output stream. This function will normally be used for writing *binary* files.
- `ostream& ostream::flush()`: any pending output on the output stream is written.

### 12.3.6 Good, bad, and ...: IOStream 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. Furtunately, it is possible to clear these error condition, so that a program can repair the problem, instead of having to abort.

Several condition member functions of the `fstreams` exist to manipulate or determine the states of the stream:

- `bad()`: this member function returns a non-zero value when an invalid operation has been requested, like seeking before the begin of file position.
- `eof()`: this member function returns a non-zero value when the stream has reached end of file (`EOF`).
- `fail()`: this member function returns a non-zero value when `eof()` or `bad()` returns a non-zero value.

Note that once one of these error conditions are raised, further processing of the stream is suspended. The member function `good()`, on the other hand, returns a non-zero value when there are *no* error conditions. Alternatively, the operator `'!'` could be used for that in combination with `fail()`. So `good()` and `!fail()` return identical logical values.

A subtlety is the following: Assume a stream is constructed, but not attached to an actual file. E.g., the statement `ifstream instream` creates the stream object, but doesn't assign it to a file. However, if we next check it's status through `good()` this member will return a non-zero value. The 'good' status here indicates that the stream object has been cleanly constructed. It doesn't mean the file is also open. A direct test for that can be performed by inspecting `instream.rdbuf()->is_open`. If non-zero, the stream is open.

When an error condition has occurred (i.e., `fail()` returns a non-zero value), and can be repaired, then the member function `clear()` should be called to clear the error status of the file.

### 12.3.7 Formating

While the insertion and extraction operators provide elegant ways to read information from and write information to `iostreams`, there are situations in which special formating is required. Formating may involve the control of the width of an output field or an input buffer or the form (e.g., the radix) in which a value is displayed. The functions `(v)form()` and `(v)scan()` can be used for special formating. Although these latter functions are not available in all implementations, they *are* available with the `GNU g++` run-time system.

Apart from these member functions, member functions are available for defining the precision and the way numbers are displayed. Apart from using members, *manipulators* exist for controlling the display form and the width of output and input elements. Different from member functions, manipulators are part of insertion or extraction statements.

### The (v)form() and (v)scan() members

To format information to be inserted into a stream the member `form()` is available:

```
ostream& form(const char *format, ...);
```

Note that this is a *member-function*, returning a reference to an `ostream` object. Therefore, it can be used in combination with, e.g., the insertion operator:

```
cout.form("Hello %s", "world") << endl;
```

produces a well known sentence.

The member function `form()` is the analogue of **C**'s `fprintf()` function. When variadic functions are constructed in which information must be inserted into a stream, the member function `vform()` can be used, being the analogue of `vfprintf()`.

To scan information from a stream, the member function `scan()` can be used, which is the analogue of **C**'s `fscanf()` function. Similarly to `vfscanf()`, the member function `vscan()` can be used in variadic functions.

### Manipulators: dec, hex, oct and other manipulators

The `iostream` objects maintain *format states* controlling the default formating of values. The format states can be controlled by member functions and by manipulators. Manipulators are inserted into the stream, the member functions are used by themselves.

The following manipulators are available:

- `dec, hex, oct`: These manipulators enforce the display of integral numbers in, respectively, decimal, hexadecimal and octal format. The default conversion is decimal. The conversion takes effect on information inserted into the stream after processing the manipulators. So, a statement like:

  ```
  cout << 16 << ", " << hex << 16 << ", " << oct << 16;
  ```

  will produce the output

  ```
  16, 10, 20
  ```

- `setbase(int b)`: This manipulator can be used to display integral values using the base 8, 10 or 16. It can be used instead of `oct, dec, hex` in situations where the base of integral values is parameterized.

- `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.
- `setprecision(int width)`: This manipulator can be used to set the precision in which a `float` or `double` is displayed. In order to use manipulators requiring arguments the header file `iomanip` must be included.
- `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, and it can be used with extraction, where it defines the maximum number of characters that are inserted into an array.

  For example, to insert 20 characters into `cout`, use:

  ```
  cout << setw(20) << 8 << endl;
  ```

  To prevent array-bounds overflow when extracting from `cin`, `setw()` can be used as well:

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

  A nice feature here is that a long string appearing at `cin` is split into substrings of at most `sizeof(array) - 1` characters, and an ascii-z is 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 2 or 4 bytes, is usually not the size of the array that it points to....
  - In order to use `setw()` the header file `iomanip` must be included.

### Setting the precision: the member precision()

The function `precision()` is used to define the precision of the display of floating point numbers. The function expects the number of digits (*not* counting the decimal point or the minus sign) that are to be displayed as its argument. For example,

```
cout.precision(4);
cout << sqrt(2) << endl;
cout.precision(6);
cout << -sqrt(2) << endl;
```

results in the following output:

```
1.414
-1.41421
```

When used without argument, `precision()` returns the actual precision value:

```
cout.precision(4);
cout << cout.precision() << ", " << sqrt(2) << endl;
```

Note that `precision()` is not a manipulator, but a member function. Therefore, `cout.precision()` rather than `precision()` is inserted into the stream.

### (Un)Setting display flags: the member (un)setf()

The member function `setf()` is used to define the way numbers are displayed. It expects one or two arguments, all *flags* of the `iostream` class. In the following examples, `cout` is used, but other `ostream` objects might have been used as well:

- To display the numeric base of integral values, use

  `cout.setf(ios::showbase)`

  This results in *no* prefix for decimal values, `0x` for hexadecimal values, `0` for octal values. For example:

  ```
  cout.setf(ios::showbase);
  cout << 16 << ", " << hex << 16 << ", " << oct << 16 << endl;
  ```

  results in:

  `16, 0x10, 020`

- To display a trailing decimal point and trailing decimal zeros when real numbers are displayed, use

  `cout.setf(ios::showpoint)`

  For example:

  ```
  cout.setf(ios::showpoint);
  cout << 16.0 << ", " << 16.1 << ", " << 16 << endl;
  ```

  results 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.

  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.

- Comparable to the `dec, hex` and `oct` manipulators

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

  or

  ```
  cout.setf(ios::oct, ios::basefield);
  ```

  can be used.

- To control the way real numbers are displayed `cout.setf(ios::fixed, ios::floatfield)` or `cout.setf(ios::scientific, ios::floatfield)` can be used. These settings result in, respectively, a fixed value display or a scientific (power of 10) display of numbers. For example,

  ```
  cout.setf(ios::fixed, ios::floatfield);
  cout << sqrt(200) << endl;
  cout.setf(ios::scientific, ios::floatfield);
  cout << sqrt(200) << endl;
  ```

  results in

  ```
  14.142136
  1.414214e+01
  ```

- `ios::left`: This format state is used to left-adjust the display of values for which the `setw()` manipulator (see below) is used. The format state can be set using the `setf()` member function, and it can be unset using the `unsetf()` member function. By default values are right-adjusted.

- `ios::internal`: This format state will add the fill-characters (blanks by default) between the minus sign of negative numbers and the value itself.

With `istream` objects the flag `ios::skipws` can be used to control the handling of whitespace characters when characters are extracted. Leading white space characters of numerical values are skipped when `istreamObject.unsetf(ios::skipws)` has been specified, but otherwise they must be read explicitly. Reading a `char` $*$ or `string` variable in this situation will only succeed if the first character to be read isn't a white-space character. The following small program can be used to illustrate the effects of unsetting `ios::skipws`:

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

```
int main()
{
    string
        buffer;
    int
        i;
    char
        c;

    cin.unsetf(ios::skipws);

    cin >> i;           // skips leading ws
    cin >> buffer;      // doesn't skip leading ws.

    cout << "got " << i << " and " << buffer << endl;

    while (cin >> c)    // reads individual chars, if the previous
        cout << "got '" << c << "'\n";    // extraction succeeded.

    return (0);
}
```

Summarizing:

- `setf(ios::showbase)` is used to display the numeric base of integral values,
- `setf(ios::showpoint)` is used to display the trailing decimal point and trailing zeros of real numbers
- `setf(ios::dec, ios::basefield)`, `setf(ios::hex, ios::basefield)` and `setf(ios::oct, ios::basefield)` can be used instead of the `dec`, `hex` and `oct` manipulators.
- `cout.setf(ios::scientific, ios::floatfield)` and `cout.setf(ios::fixed, ios::floatfield)` can be used to obtain a fixed or scientific (power of 10) display of real values.
- `setf(ios::left)` is used to left-adjust values in the width of their fields
- `setf(ios::internal)` is used to left-adjust the minus sign of negative values (while the values themselves are right adjusted).
- `ios::skipws` is used to control the handling of white space characters by the extraction operator.

To *unset* flags, the function `unsetf()` can be used.

### 12.3.8 Constructing manipulators

Using a construction like `cout << hex << 13 << endl` the value 13 is displayed 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.

Furthermore, it must be possible to use the manipulator in a <<-sequence. This implies that the return value of the manipulator must be a reference to an `ostream` object also.

From the above considerations we're now able to construct our `w10` function:

```
#include <iostream>
#include <iomanip>

ostream &w10(ostream &str)
{
    return (str << setw(10));
}
```

The `w10` function can of course be used in a 'stand alone' mode, but it can also be used as a manipulator. E.g.,

```
#include <iostream>
#include <iomanip>

extern ostream &w10(ostream &str);

int main()
{
    w10(cout) << 3 << " ships sailed to America" << endl;
    cout << "And " << w10 << 3 << " other ships sailed too." << endl;
}
```

The `w10` function can be used as manipulator because the `class ostream` has an overloaded `operator`<< accepting a pointer to a function that takes an `ostream &` and returns an `ostream &`. Its definition is:

```
ostream& operator<<(ostream & (*func)(ostream &str))
{
    return ((*func)(*this));
}
```

# Chapter 13

# Exceptions

In **C** there are several ways to have a program react to situations which break the normal unhampered flow of the program:

- The function may notice the abnormality and issue a message. This is probably the least disastrous reaction a program may show.
- The function in which the abnormality is observed may decide to stop its intended task, returning an errorcode 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 proceed at an outer section, skipping the intermediate levels which would have to be visited if a series of returns from nested functions would have been used.

In **C++** all the above ways to handle flow-breaking situations are still available. However, the last way, using `setjmp()` and `longjmp()` isn't often seen in **C++** (or even in **C**) programs, due to the fact that the program flow is completely disrupted.

In **C++** the alternative to using `setjmp()` and `longjmp()` are *exceptions*. Exceptions are a mechanism by which a controlled non-local exit is realized within the context of a **C++** program, without the disadvantages of `longjmp()` and `setjmp()`.

Exceptions are the proper way to bail out of a situation which cannot be handled easily by a function itself, but which are not disastrous enough for the program to terminate completely. Also, exceptions provide a flexible layer of flow control between the short-range `return` and the crude `exit()`.

In this chapter the use of exceptions and their syntax will be discussed. First an example of the different impacts exceptions and `setjmp()` and `longjmp()` have on the program will be given. Then the discussion will dig into the formalities of the use of exceptions.

## 13.1 Using exceptions: an outline

Using exceptions, it involves the following syntactical elements:

- `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 expressionvalue as an exception. The `throw` statement should 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 processed
}
```

## 13.2 An example using exceptions

In the next two sections the same basic program will be used. The program uses two classes, `Outer` and `Inner`. An `Outer` object is created in the `main()` function, and the function `Outer::fun()` is called. Then, in the `Outer::fun()` function an `Inner` object is allocated. After allocating the `Inner` object, its member function `fun()` is called.

That's about it. The function `Outer::fun()` terminates, and the destructor of the `Inner` object is called. Then the program terminates and the destructor of the `Outer` object is called.

Here is the basic program:

```
#include <iostream.h>

class Inner
```

```
{
    public:
        Inner();
        ~Inner();
        void fun();
};

class Outer
{
    public:
        Outer();
        ~Outer();
        void fun();
    private:
};

Inner::Inner()
{
    cout << "Inner constructor\n";
}

Inner::~Inner()
{
    cout << "Inner destructor\n";
}

void Inner::fun()
{
    cout << "Inner fun\n";
}

Outer::Outer()
{
    cout << "Outer constructor\n";
}

Outer::~Outer()
{
    cout << "Outer destructor\n";
}

void Outer::fun()
{
    Inner
        in;

    cout << "Outer fun\n";
    in.fun();
}

int main()
{
    Outer
```

```
        out;

    out.fun();
}
```

This program can be compiled and run, producing the following output:

```
    Outer constructor
    Inner constructor
    Outer fun
    Inner fun
    Inner destructor
    Outer destructor
```

This output is completely as expected, and it is exactly what we want: the destructors are called in their correct order, reversing the calling sequence of the constructors.

Now let's focus our attention on two variants, in which we simulate a non-fatal disastrous event to take place in the `Inner::fun()` function, which is supposedly handled somewhere at the end of the function `main()`. We'll consider two variants. The first variant will try to handle this situation using `setjmp()` and `longjmp()`, the second variant will try to handle this situation using **C++**'s exception mechanism.

### 13.2.1 No exceptions: the setjmp() and longjmp() approach

In order to use `setjmp()` and `longjmp()` the basic program from section 13.2 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 returnvalue 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 returnvalue of the program is zero *if only* we would have been 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 returnvalue. 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 returnvalue 1.

Now try to follow these steps by studying the next program source, modified after the basic program given in section 13.2:

```
#include <iostream.h>
#include <setjmp.h>
```

```
#include <stdlib.h>

class Inner
{
    public:
        Inner();
        ~Inner();
        void fun();
};

class Outer
{
    public:
        Outer();
        ~Outer();
        void fun();
};

jmp_buf
    jmpBuf;

Inner::Inner()
{
    cout << "Inner constructor\n";
}

void Inner::fun()
{
    cout << "Inner fun()\n";
    longjmp(jmpBuf, 0);
}

Inner::~Inner()
{
    cout << "Inner destructor\n";
}

Outer::Outer()
{
    cout << "Outer constructor\n";
}

Outer::~Outer()
{
    cout << "Outer destructor\n";
}

void Outer::fun()
{
    Inner
        in;
    cout << "Outer fun\n";
    in.fun();
```

```
}

int main()
{
    Outer
        out;

    if (!setjmp(jmpBuf))
    {
        out.fun();
        return (0);
    }

    return (1);
}
```

Running the above program produces the following output:

```
    Outer constructor
    Inner constructor
    Outer fun
    Inner fun()
    Outer destructor
```

As will be clear from this output, the destructor of the class `Inner` is not executed. This is a direct result of the non-local characteristic of the call to `longjmp()`: from the function `Inner::fun()` processing continues immediately in the function `setjmp()` in `main()`: the call to `Inner::~Inner()`, hiddenly placed at the end of `Outer::fun()` is never executed.

Since the destructors of objects can easily be skipped when `longjmp()` and `setjmp()` are used, it's probably best to skip these function completely in **C++** program.

### 13.2.2 Exceptions: the preferred alternative

In **C++** *exceptions* are the best alternative to using `setjmp()` and `longjmp()`. In this section an example using exceptions is presented. Again, the program is derived from the basic program, given in section 13.2. The syntax of exceptions will be covered shortly, so please skip over the syntactical peculiarities like `throw, try` and `catch`. Here comes the sourcetext:

```
#include <iostream.h>
```

```
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 (...)
    {}
}
```

In this program an *exception* is thrown, where a `longjmp()` was used in the program in section 13.2.1. 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, like section 13.2.1, the execution of function `Inner::fun()` terminates, albeit with an exception, rather than a `longjmp()`. The exception is caught in `main()`, and the program terminates.

Now look at the output generated by *this* program:

```
    Outer constructor
    Inner constructor
    Outer fun
    Inner fun
    Inner destructor
    Outer destructor
```

Note that the destructor of the `Inner` object, created in `Outer::fun()` is now called again. On the other hand, execution of the function `Inner::fun()` really terminates at the `throw` statement: the insertion of the text into `cout`, just beyond the `throw` statement, isn't performed.

So, with our illustrations we hope to have raised your appetite for exceptions by showing that

- Exceptions provide a means to break out of the normal flow control without having to use a cascade of `return`-statements, and without having to terminate the program.
- Exceptions do not disrupt the activation of destructors, and are therefore strongly preferred over the use of `setjmp()` and `longjmp()`.

## 13.3 Throwing exceptions

Exceptions may be generated in a `throw` statement. The `throw` keyword is followed by an expression, which results in a value of a certain type. For example:

```
    throw "Hello world";        // throws a char *
    throw 18;                   // throws an int
    throw string("hello");      // throws a string
```

Objects defined locally in functions are automatically destroyed once exceptions are thrown within these functions. However, if the object itself is thrown, the exception catcher receives a copy of the thrown object. This copy is constructed just before the local object is destroyed.

The next source 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.h>
#include <string>

class Object
{
    public:
        Object(string name)
        :
            name(name)
        {
            cout << "Object constructor of " << name << "\n";
        }
        Object(Object const &other)
        :
            name(other.name + " (copy)")
        {
            cout << "Copy constructor for " << name << "\n";
        }
        ~Object()
        {
            cout << "Object destructor of " << name << "\n";
        }
        void fun()
        {
            Object
                toThrow("'local object'");

            cout << "Object fun() of " << name << "\n";
            throw toThrow;
        }
        void hello()
        {
            cout << "Hello by " << name << "\n";
        }
    private:
        string
            name;
```

```
};

int main()
{
    Object
        out("'main object'");

    try
    {
        out.fun();
    }
    catch (Object o)
    {
        cout << "Caught exception\n";
        o.hello();
    }
}
```

The class `Object` defines some simple constructors and members. The copy constructor is special in that it adds the text `" (copy)"` to the received name, to allow us to monitor the construction and destruction of objects somewhat more closely. The member function **`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'
```

Then 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. Next, the exception is processed. The local object is destroyed, and the catcher catches an `Object`, which again is a value parameter. Hence, another copy is created. We see the following lines:

```
Object destructor of 'local object'
Copy constructor for 'local object' (copy) (copy)
```

Now the message inside the catcher is displayed, and the hello member of the object received by the catcher is called, showing us once again that we received a copy of the copy of the local object of the `fun()` member function:

```
Caught exception
Hello by 'local object' (copy) (copy)
```

Now the program terminates, and the still living objects are 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 implemented so as to receive a reference to an object (`catch (Object &o)`), the double copy would have been avoided. In that case the output of the program would have been:

```
Object constructor of 'main object'
Object constructor of 'local object'
Object fun() of 'main object'
Copy constructor for 'local object' (copy)
Object destructor of 'local object'
Caught exception
Hello by 'local object' (copy)
Object destructor of 'local object' (copy)
Object destructor of 'main object'
```

showing that only a single copy of the local object is 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, taking care that the generated object is properly deleted when the generated exception is caught.

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 syntax 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. A bit more sophistication would allow the use of variables, and non-existing variables may be referred to.

Each of these situations are enough reason to terminate the processing of the expression at hand, but there's no need to terminate the program. Each component of the processing of the expression may therefore throw an exception. E.g.,

```
...
if (parse(expressionBuffer)) // parsing failed ?
    throw "Syntax error in expression";
...
if (lookup(variableName))
    throw "Variable not defined";
...
if (illegalDivision())
    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 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);
```

### 13.3.1 The empty throw statement

Situations may arise in which it is required to inspect a thrown exception. Depending on the nature of the received exception, the program may continue its normal operation, or a serious event took place, requiring a more drastic reaction by the program. In a server-client situation the client may enter requests to the server in a queue. Every request placed in the queue is normally answered by the server, telling the client that the request was successfully completed, or that some sort of error has occurred. Actually, the server may have died, and the client should be able to discover this calamity, by not waiting indefinitely for the server to reply.

In this situation an intermediate exception handler is called for. A thrown exception is first inspected at the middle level. If possible it's processed there. If it's not possible to process the exception at the middle level, it's 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 able to process that particular type of exception.

In our server-client situation a function

```
initialExceptionHandler(char *exception)
```

could be designed to do so. The received message is inspected. If it's a simple message it's processed, otherwise the exception is passed on to an outer level. The implementation of `initialExceptionHandler()` shows the empty `throw` statement:

```
void initialExceptionHandler(char *exception)
{
    if (plainMessage(exception))
        handleTheMessage(exception);
    else
        throw;
}
```

As we will see below (section 13.5), 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? Suppose we have a class `Exception`, containing a member function `Exception::Type 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 `toString()`, telling us a bit more about the nature of the generated exception. This smells a lot like polymorphism, showing `process()` as a virtual function for the derived classes `Message, Warning, Mistake, Error` and `Fatal`.

Now the program may throw all these five types of exceptions Let's assume that the `Message` and `Warning` exceptions are processable by our `initialExceptionHandler()`. Then its code would become:

```
void initialExceptionHandler(Exception *e)
{
                // show the plain-text information
    cout << e->toString() << endl;

                        // Can we process it ?
    if (e->severity <= Exception::Warning)
        e->process();   // It's either a message
                        // or a warning
    else
        throw;          // No, pass it on
}
```

Due to polymorphism, `e->process()` will either process a `Message` or a `Warning`. Thrown exceptions are generated as follows:

```
throw new Message(<arguments>);
```

```
throw new Warning(<arguments>);
throw new Mistake(<arguments>);
throw new Error(<arguments>);
throw new Fatal(<arguments>);
```

All of these exceptions are processable by our `initialExceptionHandler()`, which may decide to pass exceptions upward for further processing or to process exceptions itself.

## 13.4 The try block

The `try`-block surrounds statements in which exceptions may be thrown. As we have seen, the actual `throw` statement doesn't have to be placed within the `try`-block, but may be placed in a function which is called from the `try`-block, either directly or indirectly.

The keyword `try` is followed by a set of curly braces, which acts like a standard **C++** compound statement: multiple statements and variable definitions may be placed here.

It is possible (and very common) to create *levels* in which exceptions may be thrown. For example, code within the `main()` function is surrounded by a `try`-block, forming an outer level in which exceptions can be handled. Within `main()`'s `try`-block, functions are called which may also contain `try`-blocks, forming the next level in which exceptions may be placed. As we have seen (in section 13.3.1) 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 process (uncaught) exceptions is used, which is usually to abort the program. Try to compile and run the following tiny program, and see what happens:

```
int main()
{
    throw "hello";
}
```

## 13.5 Catching exceptions

The `catch`-block contains code that is executed when an exception is thrown. Since *expressions* are thrown, the `catch`-block should know what kind of exceptions it should handle. Therefore, the keyword `catch` is followed by a parameter list having one parameter, which is of the type of the expression of the thrown exception.

So, an exception handler for `char` $*$ exceptions will have the following form:

```
catch (char const *message)
{
```

```
        // code to handle the message
    }
```

Earlier (section 13.3) we've seen that such a message doesn't have to be thrown as static string. It's also possible for a function to return a string, which is then thrown as an exception. However, if such a function creates the string to be thrown as an exception *dynamically*, the exception handler will normally have to delete the allocated memory lest memory leaks away.

Generally close attention must 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 upwards to an outer level exception handler, the received exception should *not* be deleted by the inner level handler.

Different exception types 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, the exceptions appearing at the end of a `try`-block may be of different types. In order to catch all the types that may appear at the end of a `try`-block, multiple exception handlers (i.e., `catch`-blocks) may follow the `try`-block.

The *order* in which the exception handlers are placed is important. When an exception is thrown, the first exception handler matching the type of the thrown exception is selected, remaining exception handlers are skipped. So only one exception handler following a `try`-block will be executed. Consequently, exception handlers should be placed from the ones having the most specific parameters to the ones having more general parameters. For example, if exception handlers are defined for `char` ∗s and `void` ∗\s (i.e., any old pointer) then the exception handler for the former exception type should be placed before the exception handler for the latter type:

```
    try
    {
        // code may throw char pointers
        // and other pointers
    }
    catch (char *message)
    {
        // code processing the char pointers
        // thrown as exceptions
    }
    catch (void *whatever)
    {
        // code processing all other pointers
        // thrown as exceptions
    }
```

An alternative to construct different types of exception handlers for different types of situations, it is of course also possible to design a specific class whose objects contain information about the reason for the exception. Such an approach was discussed earlier, in section 13.3.1. Using this approach, there's only one handler required, since we *know* we won't throw other types of exceptions:

```
try
{
    // code may throw only
    // Exception pointers
}
catch (Exception *e)
{
    // code processing the Exception pointer
    delete e;
}
```

The use of the `delete e` statement in the above code indicates that the `Exception` object which could be thrown as an exception in the `try`-block was created dynamically.

When the code of an exception handler that is placed beyond a `try`-block has been processed, the execution of the program continues beyond the last exception handler following that `try`-block (unless the handler uses `return, throw` or `exit()` to leave the function prematurely). So we have the following cases:

- If *no* exception was thrown within the `try`-block no exception handler is activated, and the execution continues from the last statement in the `try`-block to the first statement beyond the last `catch`-block.
- If an exception *was* thrown within the `try`-block but neither the current level nor an other level contains an appropriate exception handler, the program's default exception handler is called, usually aborting the program.
- If an exception was thrown within the `try`-block and an appropriate exception handler is available, then that the code of that exception handler is exectuted. Following the execution of the code of the exception handler, the execution of the program continues at the first statement beyond the last `catch`-block.

In all cases a `throw`-statement will result in skipping all remaining statements of the `try`-block in which the exception was thrown. However, destructors of objects defined locally in the `try`-block *are* called, and they are called before any exception handler's code is executed.

The actual construction of the `Exception` object may be performed in various degrees of sophistication. Possibilities are using a plain `new` operator, using static member functions of the class `Exception` dedicated to a particular kind of exception, returning a pointer to an `Exception` object, or using objects of classes derived from the `class Exception`, possibly involving polymorphism.

### 13.5.1 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 to be processed, all other exceptions are treated differently, e.g., they are passed on to an outer level of exception handling.

This situation is implemented using the default exception handler, which will (because of the reason given in the previous section 13.5) be placed beyond all other, more specific exception handlers.

Often the default exception handler will be used in combination with the empty throw statement, discused in section 13.3.1.

Here is an example showing the use of a default exception handler:

```
try
{
    // this code may throw
    // different types of
    // exceptions
}
catch (char *message)
{
    // code to process
    // char pointers
}
catch (int value)
{
    // code to process
    // ints
}
catch (...)
{
    // code to process other exceptions,
    // often passing the exception on
    // to outer level exception handlers:

    throw;
}
```

The reason for passing unspecified exceptions on to outer level exception handlers is simply the fact that they are unspecified: how would you process an exception if you don't know its type? In these situations the outer level exception handlers should of course know what exceptions other than `char` $*$s and `int`s to expect....

## 13.6 Declaring exception throwers

Functions that are defined elsewhere may be linked to code using those functions. These functions are normally declared in header files, either as stand-alone functions or as member-functions of a class.

These external function may of course throw exceptions. The declaration of such functions may contain a *function throw list*, in which the types of the exceptions that can be thrown by the function are specified. For example, a function that may throw `char *` and `int` exceptions can be declared as

```
void exceptionThrower() throw(char *, int);
```

A function for which a function throw list was specified is not allowed to throw other types of exceptions. A *run-time* error occurs if it does throw other types of exceptions than mentioned in the function throw list.

If a function throw list is specified in the declaration, it must also be given in the definition of the function. For example, using declaration *and* definition in the same example:

```
#include <iostream>

void intThrower() throw(int);
void charP_IntThrower() throw (char *, int);

void intThrower(int x) throw (int)
{
    if (x)
        throw x;
}

void charP_IntThrower() throw (char *, int)
{
    int
        x;
    cout << "Enter an int: ";
    cout.flush();
    cin >> x;

    intThrower(x);
    throw "from charP_IntThrower() with love";

}

int main()
{
    try
    {
         charP_IntThrower();
    }
    catch (char *message)
    {
        cout << "Text exception: " << message << endl;
    }
    catch (int value)
    {
        cout << "Int exception: " << value << endl;
    }
    return (0);
}
```

In the function `charP_IntThrower()` the `throw` statement clearly throws a `char *`. However, since `IntThrower()` may throw an `int` exception, the function throw list of `charP_IntThrower()` must *also* contain `int`. Try this: remove the `int` from the (two!) function throw lists, compile and link the program and see what happens if you enter the value 5.

If a function *doesn't* throw exceptions an empty function throw list may be used. E.g.,

```
void noExceptions() throw ();
```

Again, the function definition must also contain the empty function throw list in this case.

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 all responsibilities of providing the correct handlers is in the hands of the designer of the program....

# Chapter 14

# More about friends

Let's return to `friend`s once more. In section 5.6 the possibility of declaring a function or class as a *friend* of a class was discussed. At the end of that section, we mentioned

- Friendship, when applied to program design, is an *escape mechanism* which circumvents the principle of data hiding. Using friend classes should therefore be *minimized* to those cases where it is absolutely essential.
- If friends are used, realize that the implementation of classes or functions that are friends to other classes become implementation dependent on these classes. In the above example: once the internal organization of the data of the class `A` changes, all its friends must be recompiled (and possibly modified) as well.
- As a rule of thumb: *don't* use friend functions or classes.

In our opinion, there are indeed very few reasons for using the `friend` keyword. It violates the principle of data hiding, and makes the maintenance of a class dependent on another class.

Nonetheless, it might be worthwhile to look at some examples in which the `friend` keyword can be used profitably. Having seen such examples, the decision about whether or not to use `friend`s might be based on a somewhat more solid foundation than on a plain rule of thumb.

At the onset, we remark that in our programming projects we *never* found any convincing reason to resort to `friend`s. Having thus made our position clear, 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 an old class access to a class developed *later* in history.

However, while developing the older class, it was not yet known that the newer class would be developed later in time. E.g., the older class is distributed in the runtime-library of a compiler, and the newer class is a class developed by us.

Consequently, no provisions were offered in the older class to access the information in the newer class.

Consider the following situation. Within the `C++` I/O-library the extraction $>>$ and insertion $<<$ operators may be used to extract from and to insert into a stream.

These operators can be given data of several types: `int, double, char *`, etc.. Now assume that we develop a class `String`. Objects of the `class String` can be given a string, and `String` objects can also *produce* other `String` objects.

While it is possible to use the insertion operator to write the string that is stored in the object to a stream, it is not possible to use the *extraction* operator, as illustrated by the following piece of code:

```
#include <iostream>

class String
{
    public:
        // ...
        void set(char const *s);
        char const *get() const;
    private:
        char
            *str;
};

void f()
{
    String
        str;

    str.set("hello world");
        // Assign a value. Can't use
        // cin >> str.set() or
        // a similar construction

    cout << str.get() << endl;
        // this is ok.
}
```

Actually, the use of the insertion operator in combination with the `String` class is also a bit of a kludge: it isn't the `String` object that is inserted into the stream, but rather a string produced by one of its members.

Below we'll discuss a method to allow the insertion and extraction of `String` objects, based on the use of the `friend` keyword.

## 14.1 Inserting String objects into streams

Assume that we would like to be able to insert `String` objects into streams, rather than derivatives of `String` objects, like `char const *`'s. If we would be able to write `String` objects into streams, we could be using code comparable to

```
int main()
{
    String
        str("Hello world");

    cout << "The string is: '" << str << "'" << endl;
    return (0);
}
```

Analogously, with the extraction operator, we would like to be able to write code comparable to the next example:

```
int main()
{
    String
        str;

    cout << "Enter your string: ";

    cin >> str;

    cout << "Got: '" << str << "'" << endl;

    return (0);
}
```

In this situation we would not have to rely on the availability of a particular member (like `char const *String::get()`), and we would be able to fill a `String` object directly via the extraction operator, rather than via an intermediate variable of a type understood by the `cin` stream.

Even more central to the concept of object oriented programming: we would be able to *ignore* the functionality of the `String` class in combination with `iostream` objects: our objective is, after all, to insert the information in the `String` *object* into the `cout` stream, and not to call a particular function to do so.

Once we're able to focus our attention on the object, rather than on its member functions, the above piece of code remains valid, no matter what internal organization the `String` object has.

## 14.2 An initial solution

Consider the following overloaded operator >>, to be used as an extraction operator with a `String` object:

```
istream &String::operator>>(istream &is)
{
```

```
    char
        buffer[500];
        // assume this buffer to be
        // large enough.

    is >> buffer;      // extraction

    delete str;     // free this->str
                    // memory

                    // assign new value
    str = strdupnew(buffer);

    return (is);    // return is-reference
}
```

The extraction operator can now be used with `String` objects. Unfortunately, this implementation produces awkward code. The extraction operator is part of the `String` class, so its *left* operand must be a `String` object.

As the *left* operand must be a `String` object, we're now forced to use weird-looking code like the following, which can only partially be compiled. The numbered statements are annotated next.

```
void fun()
{
    String
        s;

    s >> cin;           // (1)

    int x;

    s >> (cin >> x);    // (2)

    cin >> x >> s;      // (3)
}
```

1. In this statement `s` is the *left-hand* operator, and `cin` the *right-hand*, consequently, this statement represents *extraction from a* `cin` *object into a* `String` *object.*
2. In this statement parentheses are needed to indicate the proper ordering of the sub-expressions: first `cin` >> `x` is executed, producing an `istream &`, which is then used as a right-hand operand with the `extraction` to `s`.
3. This statement is what we want, but it doesn't compile: the `istream`'s overloaded operator >> doesn't know how to extract information into `String` objects.

## 14.3 Friend-functions

The last statement of the previous example is in fact what we want. How can we accomplish the syntactical (and semantical) correctness of that last statement?

A solution is to overload the *global* >> operator to accept a left-operand of the `istream &` type, and a right operand of the `String &` type, returning an `istream &`. Its prototype is, therefore:

```
istream &operator>>(istream &is, String &destination);
```

To implement this function, the implementation given for the overloaded extraction operator of the `String` class can't simply be copied, since the private datamember `str` is accessed there. A small (and perfectly legal) modification would be to access the `String`'s information via a `char const *String::get() const` member, but this would again generate a dependency on the `String::get()` function, which we would like to avoid.

However, the need for overloading the extraction operator arose strictly in the context of the `String` class, and is in fact depending on the existence of that class. In this situation the overloading of the operator could be considered an extension to the `String` class, rather than to the `iostream` class.

Next, since we consider the overloading of the >> operator in the context of the `String` class an *extension of* the `String` class, we feel safe to allow that function access to the private members of a `String` object, instead of forcing the `operator>>()` function to assign the data members of the `String` object through the `String`'s member functions.

Access to the private data members of the `String` object is granted by declaring the `operator>>()` function to be a `friend` of the `String` class:

```
#include <iostream>

class String
{
    friend istream &operator>>(istream &is,
                               String &destination);
    public:
        // ...
    private:
        char
            *str;
};

istream &operator>>(istream &is, String &destination)
{
    char
        buffer[500];

    is >> buffer;                  // extraction

    delete destination.str;     // free old 'str' memory

    destination.str = strdupnew(buffer);
```

```
                                // assign new value

        return (is);                // return istream-reference
    }

    void fun()
    {
        String
            s;

        cin >> s;   // application

        int
            x;

        cin >> x >> s;
                    // extraction order is now
                    // as expected
    }
```

Note that nothing in the implementation of the `operator>>()` function suggests that it's a friend of the `String` class. The compiler detects this only from the `String` interface, where the `operator>>()` function is declared as a friend.

### 14.3.1 Preventing the friend-keyword

Now that we've seen that it's possible to define an overloaded `operator>>()` function for the `String` class, it's hopefully clear that there is only very little reason to declare it as a friend of the `class String`, assuming that the proper member functions of the class are available.

On the other hand, declaring the `operator>>()` as a friend function isn't that much of a problem, as the `operator>>()` function can very well be interpreted as a true member function of the `class String`, although, due to a syntactical peculiarity, it cannot be defined as such.

To illustrate the possibility of overloading the >> operator for the `istream` and `String` combination, we present here the version which does not have to be declared as a `friend` in the `String` class interface. This implementation assumes that the class `String` has an overloaded operator `=`, accepting as r-value a `char const *`:

```
    istream &operator>>(istream &lvalue, String &rvalue)
    {
        char
            buffer[500];

        lvalue >> buffer;       // extraction

        rvalue = buffer;        // assignment

        return (lvalue);    // return istream-reference
    }
```

No big deal, isn't it? After all, whether or not to use `friend` functions might purely be a matter of taste. As yet, we haven't come across a situation where `friend` functions are truly needed.

## 14.4 Friend classes

Situations may arise in which two classes doing closely related tasks are developed together.

For example, a window application can define a class `Window` to contain the information of a particular window, and a class `Screen` shadowing the `Window` objects for those windows that are actually visible on the screen.

Assuming that the window-contents of a `Window` or `Screen` object are accessible through a `char *win` pointer, of `unsigned size` characters, an overloaded operator `!=` can be defined in one (or both) classes to compare the contents of a `Screen` and `Window` object immediately. Objects of the two classes may then be compared directly, as in the following code fragment:

```
void fun()
{
    Screen
        s;
    Window
        w;

    // ... actions on s and w ...

    if (w != s)         // refresh the screen
        w.refresh(s);   // if w != s
}
```

It is likely that the overloaded operator `!=` and other member functions of `w` (like `refresh()`) will benefit from direct access to the data of a `Screen` object. In this case the class `Screen` may declare the class `Window` as a friend class, thus allowing `Window`'s member functions to access the private members of its objects.

A (partial) implementation of this situation is:

```
class Window;       // forward declaration
class Screen
{
    friend class Window;    // Window's object may
                            // access Screen's
                            // private members
    public:
        // ...
    private:
```

```
        // ...
        char
            *win;
        unsigned
            size;
};

// ============================================
// now in Window's context:

int Window::operator!=(Screen const &s)
{
    return
    (
        s.size != size      // accessing Screen's
        ||                  // private members
        !memcmp(win, s.win, size)
    );
};
```

It is also possible to declare classes to be each other's friends, or to declare a global function to be a friend in multiple classes. While there may be situations where this is a useful thing to do, it is important to realize that these multiple friendships actually violate the principle of encapsulation.

In the example we've been giving earlier for single `friend` functions, the implementation of such functions can be placed in the same directory as the actual member functions of the class declaring the function to be its friend. Such functions can very well be considered part of the class implementation, being somewhat 'eccentric' member functions. Those functions will normally be inspected automatically when the implementation of the data of the class is changed.

However, when a class itself is declared as a friend of another class, things become a little more complex. If the sources of classes are kept and maintained in different directories, it is not clear where the code of `Window::operator!=()` should be stored, as this function accesses private members of both the class `Window` and `Screen`. Consequently caution should be exercized when these situations arise.

In our opinion it's probably best to avoid friend classes, as they violate of the central principle of encapsulation.

# Chapter 15

# Inheritance

When programming in **C**, it is common to view problem solutions from a *top-down* approach: functions and actions of the program are defined in terms of sub-functions, which again are defined in sub-sub-functions, etc.. This yields a hierarchy of code: `main()` at the top, followed by a level of functions which are called from `main()`, etc..

In **C++** the dependencies between code and data can also be defined in terms of *classes* which are *related to other classes*. This looks like *composition* (see section 5.5), where objects of a class contain objects of another class as their data. But the relation which is described here is of a different kind: a class can be *defined* by means of an older, pre-existing, class. This leads to a situation in which a new class has all the functionality of the older class, and additionally introduces its own specific functionality. Instead of composition, where a given class *contains* another class, we mean here *derivation*, where a given class *is* another class.

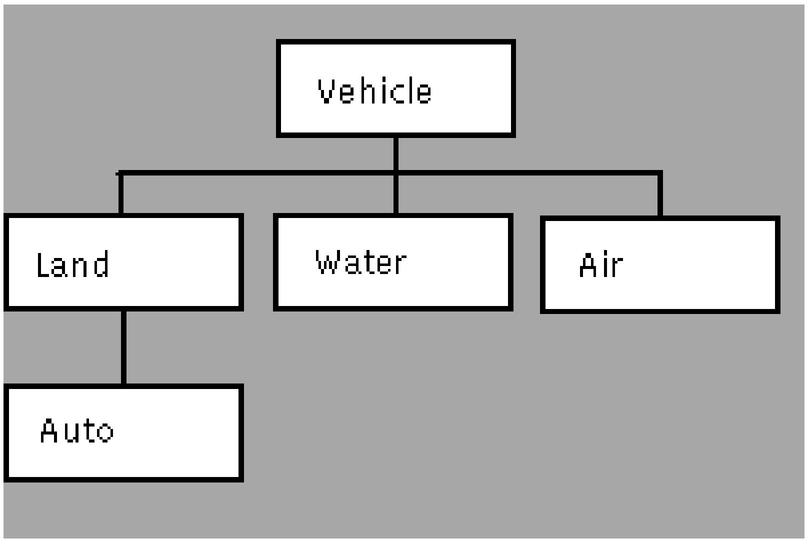
Another term for derivation is *inheritance*: the new class inherits the functionality of an existing class, while the existing class does not appear as a data member in the definition of the new class. When speaking of inheritance the existing class is called the *base class*, while the new class is called the *derived class*.

Derivation of classes is often used when the methodology of **C++** program development is fully exploited. In this chapter we will first address the syntactical possibilities which **C++** offers to derive classes from other classes. Then we will address the peculiar extension to **C** which is thus offered by **C++**.

As we have seen the object-oriented approach to problem solving in the introductory chapter (see section 2.4), classes are identified during the problem analysis, after which objects of the defined classes can be declared to represent entities of the problem at hand. The classes are placed in a hierarchy, where the top-level class contains the least functionality. Each derivation and hence descent in the hierarchy adds functionality in the class definition.

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 15.1.




\

Figure 15.1: Initial object hierarchy of vehicles.

## 15.1 Related types

The relationship between the proposed classes representing different kinds of vehicles is further illustrated here. The figure shows the object hierarchy in vertical direction: an `Auto` is a special case of a `Land` vehicle, which in turn is a special case of a `Vehicle`.

The class `Vehicle` is thus the 'greatest common denominator' in the classification system. For the sake of the example we implement in this class the functionality to store and retrieve the weight of a vehicle:

```
class Vehicle
{
    public:
        // constructors
        Vehicle();
        Vehicle(int wt);

        // interface
        int getweight() const;
        void setweight(int wt);

    private:
        // data
        int weight;
};
```

Using this class, the weight of a vehicle can be defined as soon as the corresponding object is created. At a later stage the weight can be re-defined or retrieved.

To represent vehicles which travel over land, a new class `Land` can be defined with the functionality of a `Vehicle`, but in addition its own specific information. For the sake of the example we assume that we are interested in the speed of land vehicles *and* in their weight. The relationship between `Vehicle`s and `Land`s could of course be represented with composition, but that would be awkward: composition would suggest that a `Land` vehicle *contains* a vehicle, while the relationship should be that the `Land` vehicle *is* a special case of a vehicle.

A relationship in terms of composition would also introduce needless code. E.g., consider the following code fragment which shows a class `Land` using composition (only the `setweight()` functionality is shown):

```
class Land
{
    public:
        void setweight(int wt);
    private:
        Vehicle v;      // composed Vehicle
};

void Land::setweight(int wt)
{
    v.setweight(wt);
}
```

Using composition, the `setweight()` function of the class `Land` would only serve to pass its argument to `Vehicle::setweight()`. Thus, as far as weight handling is concerned, `Land::setweight()` would introduce no extra functionality, just extra code. Clearly this code duplication is redundant: a `Land` should *be* a `Vehicle`, and not: a `Land` should *contain* a `Vehicle`.

The relationship is better achieved with inheritance: `Land` is *derived* from `Vehicle`, in which `Vehicle` is the base class of the derivation.

```
class Land: public Vehicle
{
    public:
        // constructors
        Land();
        Land(int wt, int sp);

        // interface
        void setspeed(int sp);
        int getspeed() const;

    private:
        // data
        int speed;
};
```

By postfixing the class name `Land` in its definition by `public Vehicle` the derivation is defined:

the class `Land` now contains all the functionality of its base class `Vehicle` plus its own specific information. The extra functionality consists here of a constructor with two arguments and interface functions to access the `speed` data member.[1].

To illustrate the use of the derived class `Land` consider the following example:

```
Land
    veh(1200, 145);

int main()
{
    cout << "Vehicle weighs " << veh.getweight() << endl
         << "Speed is " << veh.getspeed() << endl;

    return (0);
}
```

This example shows two features of derivation. First, `getweight()` is no direct member of a `Land`. Nevertheless it is used in `veh.getweight()`. This member function is an implicit part of the class, inherited from its 'parent' vehicle.

Second, although the derived class `Land` now contains the functionality of `Vehicle`, the `private` fields of `Vehicle` remain private in the sense that they can only be accessed by member functions of `Vehicle` itself. This means that the member functions of `Land` *must* use the interface functions (`getweight()`, `setweight()`) to address the `weight` field; just as any other code outside the `Vehicle` class. This restriction is necessary to enforce the principle of data hiding. The class `Vehicle` could, e.g., be recoded and recompiled, after which the program could be relinked. The class `Land` itself could remain unchanged.

Actually, the previous remark is not quite right: If the internal organization of the `Vehicle` changes, then the internal organization of the `Land` objects, containing the data of `Vehicle`, changes as well. This means that objects of the `Land` class, after changing `Vehicle`, might require more (or less) memory than before the modification. However, in such a situation we still don't have to worry about the use of member functions of the parent class `Vehicle` in the class `Land`. We might have to recompile the `Land` sources, though, as the relative locations of the data members within the `Land` objects will have changed due to the modification of the `Vehicle` class.

To play it safe, classes which are derived from other classes must be fully recompiled (but don't have to be modified) after changing the *data organization* of their base class(es). As adding new *member functions* to the base class doesn't alter the data organization, no such recompilation is needed after adding new *member functions*. (A subtle point to note, however, is that adding a new member function that happens to be the *first virtual* member function of a class results in a hidden pointer to a table of pointers to virtual functions. This topic is discussed further in chapter 16).

In the following example we assume that the class `Auto`, representing automobiles, should be able to contain the weight, speed and name of a car. This class is therefore derived from `Land`:

```
class Auto: public Land
```

[1]The derivation in this example mentions the keyword `public`. C++ also implements `private` derivation, which is not often used and which we will therefore leave to the reader to uncover.

```
{
    public:
        // constructors
        Auto();
        Auto(int wt, int sp, char const *nm);

        // copy constructor
        Auto(Auto const &other);

        // assignment
        Auto const &operator=(Auto const &other);

        // destructor
        ~Auto();

        // interface
        char const *getname() const;
        void setname(char const *nm);

    private:
        // data
        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 overloaded assignment function in the class `Auto`. Since this class uses a pointer to reach allocated memory, these tools are needed.

## 15.2 The constructor of a derived class

As mentioned earlier, a derived class inherits the functionality from its base class. In this section we shall describe the effects of the inheritance on the constructor of a derived class.

As can be seen 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 (int wt, int sp)
{
    setweight(wt);
    setspeed(sp);
}
```

This implementation has the following disadvantage. The **C++** compiler will generate code to call the default constructor of a base class from each constructor in the derived class, unless explicitly instructed otherwise. This can be compared to the situation which arises in composed objects (see section 5.5).

Consequently, in the above implementation (a) the default constructor of a `Vehicle` is called, which probably initializes the weight of the vehicle, and (b) subsequently the weight is redefined by calling `setweight()`.

A better solution is of course to call directly the constructor of `Vehicle` expecting an `int` argument. The syntax to achieve this is to mention the constructor to be called (supplied with an argument) immediately following the argument list of the constructor of the derived class itself:

```
Land::Land(int wt, int sp)
:
    Vehicle(wt)
{
    setspeed(sp);
}
```

## 15.3 The destructor of a derived class

Destructors of classes are called automatically when an object is destroyed. This rule also holds true for objects of classes that are derived from other classes. Assume we have the following situation:

```
class Base
{
    public:
        ...         // members
        ~Base();    // destructor
};

class Derived
{
    public:
        ...         // members
        ~Derived(); // destructor
}

...                 // other code

int main()
{
    Derived
        derived;

    ...
    return (0);
}
```

At the end of the `main()` function, the `derived` object ceases to exists. Hence, its destructor `Derived::~Derived()` is called. However, since `derived` is also a `Base` object, the `Base::~Base()` destructor is called as well.

It is *not* necessary to call the `Base::~Base()` destructor explicitly from the `Derived::~Derived()` destructor.

Constructors and destructors are called in a stack-like fashion: when `derived` is constructed, the appropriate `Base` constructor is called first, then the appropriate `Derived` constructor is called. When `derived` is destroyed, the `Derived` destructor is called first, and then the `Base` destructor is called for that object. In general, a derived class destructor is called before a base class destructor is called.

## 15.4 Redefining member functions

The actions of all functions which are defined in a base class (and which are therefore also available in derived classes) can be redefined. This feature is illustrated in this section.

Let's assume that the vehicle classification system should be able to represent trucks, which consist of a two parts: the front engine, which pulls a trailer. Both the front engine and the trailer have their own weights, but the `getweight()` function should return the combined weight.

The definition of a `Truck` therefore starts with the class definition, derived from `Auto` but expanded to hold one more `int` field to represent additional weight information. Here we choose to represent the weight of the front part of the truck in the `Auto` class and to store the weight of the trailer in an additional field:

```
class Truck: public Auto
{
    public:
        // constructors
        Truck();
        Truck(int engine_wt, int sp, char const *nm,
              int trailer_wt);

        // interface: to set two weight fields
        void setweight(int engine_wt, int trailer_wt);
        // and to return combined weight
        int getweight() const;

    private:
        // data
        int trailer_weight;
};

// example of constructor
Truck::Truck(int engine_wt, int sp, char const *nm,
             int trailer_wt)
```

```
:
    Auto(engine_wt, sp, nm)
{
    trailer_weight = trailer_wt;
}
```

Note that the class `Truck` now contains two functions which are already present in the base class:

- The function `setweight()` is already defined in `Auto`. The redefinition in `Truck` poses no problem: this functionality is simply redefined to perform actions which are specific to a `Truck` object.
  The definition of a new version of `setweight()` in the class `Truck` will *hide* the version of `Auto` (which is the version defined in `Vehicle`: for a `Truck` only a `setweight()` function with two `int` arguments can be used.
  However, note that the `Vehicle`'s `setweight()` function remains available. But, as the `Auto::setweight()` function is hidden it must be called explicitly when needed (e.g., inside `Truck::setweight()`. This is required even though `Auto::setweight()` has only one `int` argument, and one could argue that `Auto::setweight()` and `Truck::setweight()` are merely overloaded functions within the class `Truck`. So, the implementation of the function `Truck::setweight()` could be:

  ```
  void Truck::setweight(int engine_wt, int trailer_wt)
  {
      trailer_weight = trailer_wt;
      Auto::setweight(engine_wt);     // note: Auto:: is required
  }
  ```

- Outside of the class the `Auto`-version of `setweight()` is accessed through the scope resolution operator. So, if a `Truck t` needs to set its `Auto` weight, it must use

  ```
  t.Auto::setweight(x)
  ```

- An alternative to using the scope resolution operator is to include the base-class functions in the class interface as inline functions. This might be an elegant solution for the occasional function. E.g., if the interface of the `class Truck` contains

  ```
  void setweight(int engine_wt)
  {
      Auto::setweight(engine_wt);
  }
  ```

  then the single argument `setweight()` function can be used by `Truck` objects without using the scope resolution operator. As the function is defined inline, no overhead of an extra function call is involved.

- The function `getweight()` is also already defined in `Vehicle`, with the same argument list as in `Truck`. In this case, the class `Truck` *redefines* this member function.

  The next code fragment presents the redefined function `Truck::getweight()`:

```
int Truck::getweight() const
{
    return
        (                               // sum of:
            Auto::getweight() +         //   engine part plus
            trailer_weight              //   the trailer
        );
}
```

The following example shows the actual usage of the member functions of the class `Truck` to display several of its weights:

```
int main()
{
    Land
        veh(1200, 145);

    Truck
        lorry(3000, 120, "Juggernaut", 2500);

    lorry.Vehicle::setweight(4000);

    cout << endl << "Truck weighs " << lorry.Vehicle::getweight() << endl
        << "Truck + trailer weighs " << lorry.getweight() << endl
        << "Speed is " << lorry.getspeed() << endl
        << "Name is " << lorry.getname() << endl;
    return (0);
}
```

Note the explicit call to `Vehicle::setweight(4000)`: in order to reach the hidden member function `Vehicle::setweight()`, which *is* part of the set of member functions available to the class `Vehicle`, is must be called explicitly, using the `Vehicle::` scope resolution. As said, this is remarkable, because `Vehicle::setweight()` can very well be considered an overloaded version of `Truck::setweight()`. The situation with `Vehicle::getweight()` and `Truck::getweight()` is a different one: here the function `Truck::getweight()` is a *redefinition* of `Vehicle::getweight()`, so in order to reach `Vehicle::getweight()` a scope resolution operation (`Vehicle::`) is required.

## 15.5 Multiple inheritance

In the previously described derivations, a class was always derived from *one* base class. **C++** also implements *multiple derivation*, in which a class is derived from several base classes and hence inherits the functionality from more than one 'parent' at the same time.

For example, let's assume that a class `Engine` exists with the functionality to store information about an engine: the serial number, the power, the type of fuel, etc.:

```
class Engine
{
    public:
        // constructors and such
        Engine();
        Engine(char const *serial_nr, int power,
               char const *fuel_type);

        // tools needed as we have pointers in the class
        Engine(Engine const &other);
        Engine const &operator=(Engine const &other);

        ~Engine();

        // interface to get/set stuff
        void setserial(char const *serial_nr);
        void setpower(int power);
        void setfueltype(char const *type);

        char const *getserial() const;
        int getpower() const;
        char const *getfueltype() const;

    private:
        // data
        char const
            *serial_number,
            *fuel_type;
        int
            power;
};
```

To represent an `Auto` but with all information about the engine, a class `MotorCar` can be derived from `Auto` **and** from `Engine`, as illustrated in the below listing. By using multiple derivation, the functionality of an `Auto` *and* of an `Engine` are combined into a `MotorCar`:

```
class MotorCar
:
    public Auto,
    public Engine
{
    public:
        // constructors
        MotorCar();
        MotorCar(int wt, int sp, char const *nm,
                 char const *ser, int pow, char const *fuel);
};

MotorCar::MotorCar(int wt, int sp, char const *nm,
```

```
                    char const *ser, int pow, char const *fuel)
:
    Engine (ser, pow, fuel),
    Auto (wt, sp, nm)
{
}
```

A few remarks concerning this derivation are:

- The keyword `public` is present both before the classname `Auto` and before the classname `Engine`. This is so because the default derivation in **C++** is `private`: the keyword `public` must be repeated before each base class specification.
- The multiply derived class `MotorCar` introduces no 'extra' functionality of its own, but only combines two pre-existing types into one aggregate type. Thus, **C++** offers the possibility to simply sweep multiple simple types into one more complex type.

  This feature of **C++** is very often used. Usually it pays to develop 'simple' classes each with its strict well-defined functionality. More functionality can always be achieved by combining several small classes.
- The constructor which expects six arguments contains no code of its own. Its only purpose is to activate the constructors of the base classes. Similarly, the class definition contains no data or interface functions: here it is sufficient that all interface is inherited from the base classes.

Note also the syntax of the constructor: following the argument list, the two base class constructors are called, each supplied with the correct arguments. It is also noteworthy that the *order* in which the constructors are called is defined by the *interface*, and *not* by the implementation (i.e., by the statement in the constructor of the class `MotorCar`. This implies that:

- First, the constructor of `Auto` is called, since `MotorCar` is first of all derived from `Auto`.
- Then, the constructor of `Engine` is called,
- Last, any actions of the constructor of `MotorCar` itself are executed (in this example, none).

Lastly, it should be noted that the multiple derivation in this example may feel a bit awkward: the derivation implies that `MotorCar` *is* an `Auto` and at the same time it *is* an `Engine`. A relationship 'a `MotorCar` *has* an `Engine`' would be expressed as composition, by including an `Engine` object in the data of a `MotorCar`. But using composition, unnecessary code duplication occurs in the interface functions for an `Engine` (here we assume that a composed object `engine` of the class `Engine` exists in a `MotorCar`):

```
void MotorCar::setpower(int pow)
{
    engine.setpower(pow);
}

int MotorCar::getpower() const
```

```
{
    return (engine.getpower());
}

// etcetera, repeated for set/getserial(),
// and set/getfueltype()
```

Clearly, such simple interface functions are avoided completely by using derivation. Alternatively, when insisting on the `has` relationship and hence on composition, the interface functions could have been avoided by using `inline` functions.

## 15.6 Conversions between base classes and derived classes

When inheritance is used in the definition of classes, it can be said that an object of a derived class is at the same time an object of the base class. This has important consequences for the assignment of objects, and for the situation where pointers or references to such objects are used. Both situations will be discussed next.

### 15.6.1 Conversions in object assignments

We define two objects, one of a base class and one of a derived class:

```
Vehicle
    v(900);                 // vehicle with weight 900 kg
Auto
    a(1200, 130, "Ford");   // automobile with weight 1200 kg,
                            // max speed 130 km/h, make Ford
```

The object `a` is now initialized with its specific values. However, an `Auto` is at the same time a `Vehicle`, which makes the assignment *from* a derived object *to* a base object possible:

```
v = a;
```

The effect of this assignment is that the object `v` now receives the value 1200 as its `weight` field. A `Vehicle` has neither a `speed` nor a `name` field: these data are therefore not assigned.

The conversion from a base object to a derived object, however, is problematic: In a statement like

```
a = v;
```

it isn't clear what data to enter into the fields `speed` and `name` of the `Auto` object `a`, as they are missing in the `Vehicle` object `v`. Such an assignment is therefore not accepted by the compiler.

The following general rule applies: when assigning related objects, an assignment in which some data are dropped is legal. However, an assignment where data would have to be left blank is *not* legal. This rule is a syntactic one: it also applies when the classes in question have their overloaded assignment functions.

The conversion of an object of a base class to an object of a derived class could of course be explicitly defined using a dedicated constructor. E.g., to achieve compilability of a statement

```
a = v;
```

the class `Auto` would need an assignment function accepting a `Vehicle` as its argument. It would be the programmer's responsibility to decide what to do with the missing data:

```
Auto const &Auto::operator=(Vehicle const &veh)
{
    setweight (veh.getweight());
    .
    .  code to handle other fields should
    .  be supplied here
    .
}
```

### 15.6.2 Conversions in pointer assignments

We define the following objects and one pointer variable:

```
Land
    land(1200, 130);
Auto
    auto(500, 75, "Daf");
Truck
    truck(2600, 120, "Mercedes", 6000);
Vehicle
    *vp;
```

Subsequently we can assign `vp` to the addresses of the three objects of the derived classes:

```
vp = &land;
vp = &auto;
vp = &truck;
```

Each of these assignments is perfectly legal. However, an implicit conversion of the type of the derived class to a `Vehicle` is made, since `vp` is defined as a pointer to a `Vehicle`. Hence, when using `vp` only the member functions which manipulate the `weight` can be called, as this is the *only* functionality of a `Vehicle` and thus it is the only functionality which is available when a pointer to a `Vehicle` is used.

The same reasoning holds true for references to `Vehicles`. If, e.g., a function is defined with a `Vehicle` reference parameter, the function may be passed an object of a class that is derived from `Vehicle`. Inside the function, the specific `Vehicle` members of the object of the derived class remain accessible. This analogy between pointers and references holds true in all cases. Remember that a reference is nothing but a pointer in disguise: it mimics a plain variable, but is actually a pointer.

This restriction in functionality has furthermore an important effect for the class `Truck`. After the statement `vp = &truck`, `vp` points to a `Truck` object. Nevertheless, `vp->getweight()` will return 2600; and not 8600 (the combined weight of the cabin and of the trailer: 2600 + 6000), which would have been returned by `t.getweight()`.

When a function is called via a pointer to an object, then the *type of the pointer* and not the object itself determines which member functions are available and executed. In other words, **C++** implicitly converts the type of an object reached via a pointer to the type of the pointer pointing to the object.

There is of course a way around the implicit conversion, which is an explicit type cast:

```
Truck
    truck;
Vehicle
    *vp;

vp = &truck;        // vp now points to a truck object

Truck
    *trp;

trp = (Truck *) vp;
printf ("Make: %s\n", trp->getname());
```

The second to last statement of the code fragment above specifically casts a `Vehicle *` variable to a `Truck *` in order to assign the value to the pointer `trp`. This code will *only* work if `vp` indeed points to a `Truck` and hence a function `getname()` is available. Otherwise the program may show some unexpected behavior.

## 15.7 Storing base class pointers

The fact that pointers to a base class can be used to reach derived classes can be used to develop general-purpose classes which can process objects of the derived types. A typical example of such processing is the storage of objects, be it in an array, a list, a tree or whichever storage method may be appropriate. Classes which are designed to store objects of other classes are therefore often called *container classes*. The stored objects are *contained* in the container class.

As an example we present the class `VStorage`, which is used to store pointers to `Vehicles`. The actual pointers may be addresses of `Vehicles` themselves, but also may refer to derived types such as `Autos`.

The definition of the class is the following:

```
class VStorage
{
    public:
        VStorage();
        VSTorage(VStorage const &other);
        ~VStorage();
        VStorage const &operator=(VStorage const &other);

                            // add Vehicle& to storage
        void add(Vehicle const &vehicle);
                            // retrieve first Vehicle *
        Vehicle const *getfirst() const;
                            // retrieve next Vehicle *
        Vehicle const *getnext() const;

    private:
        // data
        Vehicle
            **storage;
        int
            nstored,
            current;
};
```

Concerning this class definition we note:

- The class contains three interface functions: one to add a `Vehicle &` to the storage, one to retrieve the first `Vehicle *` from the storage, and one to retrieve next pointers until no more are in the storage.

  An illustration of the use of this class is given in the next example:

```
Land
    land(200, 20);          // weight 200, speed 20
Auto
    auto(1200, 130, "Ford");// weight 1200 , speed 130,
                            // make Ford
VStorage
    garage;                 // the storage

garage.add(land);           // add to storage
garage.add(auto);

Vehicle const
```

```
        *anyp;
    int
        total_wt = 0;

    for (anyp = garage.getfirst(); anyp; anyp = garage.getnext())
        total_wt += anyp->getweight();

    cout << "Total weight: " << total_wt << endl;
```

This example demonstrates how derived types (one `Auto` and one `Land`) are implicitly converted to their base type (a `Vehicle &`), so that they can be stored in a `VStorage`. Base-type objects are then retrieved from the storage. The function `getweight()`, defined in the base class and the derived classes, is therupon used to compute the total weight.

- Furthermore, the class `VStorage` contains all the tools to ensure that two `VStorage` objects can be assigned to one another etc.. These tools are the overloaded assignment function and the copy constructor.
- The actual internal workings of the class only become apparent once the `private` section is seen. The class `VStorage` maintains an array of pointers to `Vehicle`s and needs two `int`s to store how many objects are in the storage and which the 'current' index is, to be returned by `getnext()`.

The class `VStorage` shall not be further elaborated; similar examples shall appear in the next chapters. It is however *very* noteworthy that by providing class derivation and base/derived conversions, **C++** presents a powerful tool: these features of **C++** allow the processing of all derived types by one generic class.

The above class `VStorage` could even be used to store all types which may be derived from a `Vehicle` in the future. It seems a bit paradoxical that the class should be able to use code which isn't even there yet, but there is no real paradox: `VStorage` uses a certain *protocol*, defined by the `Vehicle` and obligatory for all derived classes.

The above class `VStorage` has just one disadvantage: when we add a `Truck` object to a storage, then a code fragment like:

```
Vehicle const
    *any;
VStorage
    garage;

any = garage.getnext();
cout << any->getweight() << endl;
```

will *not* print the truck's combined weight of the cabin and the trailer. Only the weight stored in the `Vehicle` portion of the truck will be returned via the function `any->getweight()`. Fortunately, there is a remedy against this slight disadvantage. This remedy will be discussed in the next chapter.

# Chapter 16

# Polymorphism, late binding and virtual functions

As we have seen in the previous chapter, **C++** provides the tools to derive classes from one base type, to use base class pointers to address derived objects, and subsequently to process derived objects in a generic class.

Concerning the allowed operations on all objects in such a generic class we have seen that the base class must define the actions to be performed on all derived objects. In the example of the `Vehicle` this was the functionality to store and retrieve the weight of a vehicle.

When using a base class pointer to address an object of a derived class, the pointer type (i.e., the base class type) normally determines which function will actually be called. This means that the code example from section 15.7 using the storage class `VStorage`, will incorrectly compute the combined weight when a `Truck` object (see section 15.4) is in the storage: only one weight field of the engine part of the truck is taken into consideration. The reason for this is obvious: a `Vehicle *vp` calls the function `Vehicle::getweight()` and not `Truck::getweight()`, even when that pointer actually points to a `Truck`.

However, a remedy is available. In **C++** it is possible for a `Vehicle *vp` to call a function `Truck::getweight()` when the pointer actually points to a `Truck`.

The terminology for this feature is *polymorphism*: it is as though the pointer `vp` assumes the type of the object it points to, rather than keeping it own (base class) type. So, `vp` might behave like a `Truck *` when pointing to a `Truck`, or like an `Auto *` when pointing to an `Auto` etc..[1]

A second term for this characteristic is *late binding*. This name refers to the fact that the decision *which* function to call (a base class function or a function of a derived class) cannot be made *compile-time*, but is postponed until the program is actually executed: the right function is selected *run-time*.

[1] In one of the StarTrek movies, Cap. Kirk was in trouble, as usual. He met an extremely beautiful lady who however thereupon changed into a hideous troll. Kirk was quite surprised, but the lady told him: “Didn’t you know I am a polymorph?”

## 16.1 Virtual functions

The default behavior of the activation of a member function via a pointer is that the *type* of the pointer determines the function. 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++** with *virtual functions*.

A function becomes virtual when its declaration starts with the keyword `virtual`. Once a function is declared `virtual` in a base class, its definition remains `virtual` in all derived classes; even when the keyword `virtual` is not repeated in the definition of the derived classes.

As far as the vehicle classification system is concerned (see section 15.1 ff.) the two member functions `getweight()` and `setweight()` might be declared as `virtual`. The class definitions below illustrate the classes `Vehicle` (which is the overall base class of the classification system) and `Truck`, which has `Vehicle` as an indirect base class. The functions `getweight()` of the two classes are also shown:

```
class Vehicle
{
    public:
        Vehicle();          // constructors
        Vehicle(int wt);

                            // interface.. now virtuals!
        virtual int getweight() const;
        virtual void setweight(int wt);

    private:
        int                 // data
            weight;
}

// Vehicle's own getweight() function:
int Vehicle::getweight() const
{
    return (weight);
}

class Land: public Vehicle
{
    ...
}

class Auto: public Land
{
    ...
}

class Truck: public Auto
{
```

```
    public:
        Truck();        // constructors
        Truck(int engine_wt, int sp, char const *nm,
              int trailer_wt);

                        // interface: to set two weight fields
        void setweight(int engine_wt, int trailer_wt);
                        // and to return combined weight
        int getweight() const;

    private:
        int             // data
            trailer_weight;
};

// Truck's own getweight() function
int Truck::getweight() const
{
    return (Auto::getweight() + trailer_wt);
}
```

Note that the keyword `virtual` appears only in the definition of the base class `Vehicle`; it need not be repeated in the derived classes (though a repetition would be no error).

The effect of the late binding is illustrated in the next fragment:

```
Vehicle
    v(1200);            // vehicle with weight 1200
Truck
    t(6000, 115,        // truck with cabin weight 6000, speed 115,
      "Scania",         // make Scania, trailer weight 15000
      15000);

Vehicle
    *vp;                // generic vehicle pointer

int main()
{
    // see below (1)
    vp = &v;
    printf("%d\n", vp->getweight());

    // see below (2)
    vp = &t;
    printf("%d\n", vp->getweight());

    // see below (3)
    printf("%d\n", vp->getspeed());

    return (0);
```

```
}
```

Since the function `getweight()` is defined as `virtual`, late binding is used here: in the statements above, just below the `(1)` mark, `Vehicle`'s function `getweight()` is called. In contrast, the statements below `(2)` use `Truck`'s function `getweight()`.

Statement `(3)` however will produces a syntax error. A function `getspeed()` is no member of `Vehicle`, and hence also not callable via a `Vehicle*`.

The rule is that when using a pointer to a class, *only the functions which are members of that class can be called.* These functions can be `virtual`, but this only affects the type of binding (early vs. late).

### 16.1.1 Polymorphism in program development

When functions are defined as `virtual` in a base class (and hence in all derived classes), and when these functions are called using a pointer to the base class, the pointer as it were can assume more forms: it is polymorph. In this section we illustrate the effect of polymorphism on the manner in which programs in **C++** can be developed.

A vehicle classification system in **C** might be implemented with `Vehicle` being a union of `struct`s, and having an enumeration field to determine which actual type of vehicle is represented. A function `getweight()` would typically first determine what type of vehicle is represented, and then inspect the relevant fields:

```
enum Vtype                    // type of the vehicle
{
    is_vehicle,
    is_land,
    is_auto,
    is_truck,
}
struct Vehicle                // generic vehicle type
{
    int weight;
}
struct Land                   // land vehicle: adds speed
{
    Vehicle v;
    int speed;
}
struct Auto                   // auto: Land vehicle + name
{
    Land l;
    char *name;
}
struct Truck                  // truck: Auto + trailer
{
    Auto a;
```

```
        int trailer_wt;
    }
    union AnyVehicle            // all sorts of vehicles in 1 union
    {
        Vehicle v;
        Land l;
        Auto a;
        Truck t;
    }
    struct Object               // the data for all vehicles
    {
        Vtype type;
        AnyVehicle thing;
    }

    int getweight(Object *o)   // how to get weight of a vehicle
    {
        switch (o->type)
        {
            case is_vehicle:
                return (o->thing.v.weight);
            case is_land:
                return (o->thing.l.v.weight);
            case is_auto:
                return (o->thing.a.l.v.weight);
            case is_truck:
                return (o->thing.t.a.l.v.weight +
                        o->thing.t.trailer_wt);
        }
    }
```

A disadvantage of this approach is that the implementation cannot be easily changed. E.g., if we wanted to define a type `Airplane`, which would, e.g., add the functionality to store the number of passengers, then we'd have to re-edit and re-compile the above code.

In contrast, **C++** offers the possiblity of polymorphism. The advantage is that 'old' code remains usable. The implementation of an extra class `Airplane` would in **C++** mean one extra class, possibly with its own (virtual) functions `getweight()` and `setweight()`. A function like:

```
    void printweight(Vehicle const *any)
    {
        printf("Weight: %d\n", any->getweight());
    }
```

would still work; the function wouldn't even need to be recompiled, since late binding is in effect.

### 16.1.2 How polymorphism is implemented

This section briefly describes how polymorphism is implemented in **C++**. Understanding the implementation is not necessary for the usage of this feature of **C++**, though it does explain why there is a cost of polymorphism in terms of memory usage.

The fundamental idea of polymorphism is that the **C++** compiler does not know which function to call at compile-time; the appropriate function will be selected run-time. That means that the address of the function must be stored somewhere, to be looked up prior to the actual call. This 'somewhere' place must be accessible from the object in question. E.g., when a `Vehicle *vp` points to a `Truck` object, then `vp->getweight()` calls a member function of `Truck`; the address of this function is determined from the actual object which `vp` points to.

A common implementation is the following. An object containing virtual functions holds as its first data member a hidden field, pointing to an array of pointers holding the addresses of the virtual functions. It must be noted that this implementation is compiler-dependent, and is by no means dictated by the **C++** ANSI definition.

The table of addresses of virtual functions is shared by all objects of the class. It even may be the case that two classes share the same table. The overhead in terms of memory consumption is therefore:

- One extra pointer field per object, which points to:
- One table of pointers per (derived) class to address the virtual functions.

Consequently, a statement like `vp->getweight()` first inspects the hidden data member of the object pointed to by `vp`. In the case of the vehicle classification system, this data member points to a table of two addresses: one pointer for the function `getweight()` and one pointer for the function `setweight()`. The actual function which is called is determined from this table.

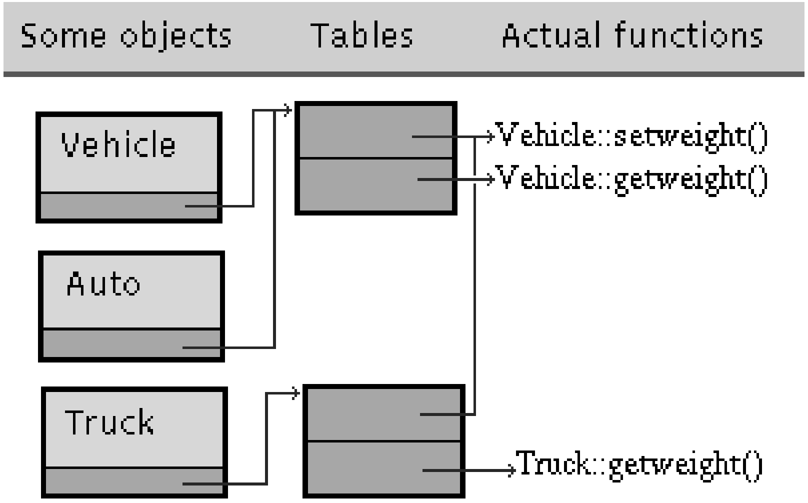
The internal organization of the objects having virtual functions is further illustrated in figure 16.1.

As can be seen from figure 16.1, all objects which use virtual functions must have one (hidden) data member to address a table of function pointers. The objects of the classes `Vehicle` and `Auto` both address the same table. The class `Truck`, however, introduces its own version of `getweight()`: therefore, this class needs its own table of function pointers.

## 16.2 Pure virtual functions

Until now the base class `Vehicle` contained its own, concrete, implementations of the virtual functions `getweight()` and `setweight()`. In **C++** it is however also possible only to *mention* virtual functions in a base class, and not define them. The functions are concretely implemented in a derived class. This approach defines a *protocol*, which has to be followed in the derived classes.

The special feature of only *declaring* functions in a base class, and not *defining* them, is that derived classes must take care of the actual definition: the **C++** compiler will not allow the definition of an object of a class which doesn't concretely define the function in question. The base class thus enforces a protocol by declaring a function by its name, return value and arguments; but the derived classes must take care of the actual implementation. The base class itself is therefore only




\

Figure 16.1: Internal organization objects when virtual functions are defined.

a *model*, to be used for the derivation of other classes. Such base classes are also called *abstract* classes.

The functions which are only declared but not defined in the base class are called *pure virtual functions*. A function is made pure virtual by preceding its declaration with the keyword `virtual` and by postfixing it with `= 0`. An example of a pure virtual function occurs in the following listing, where the definition of a class `Sortable` requires that all subsequent classes have a function `compare()`:

```
class Sortable
{
    public:
        virtual int compare(Sortable const &other) const = 0;
};
```

The function `compare()` must return an `int` and receives a reference to a second `Sortable` object. Possibly its action would be to compare the current object with the `other` one. The function is not allowed to alter the other object, as `other` is declared `const`. Furthermore, the function is not allowed to alter the current object, as the function itself is declared `const`.

The above base class can be used as a model for derived classes. As an example consider the following class `Person` (a prototype of which was introduced in chapter 6.1), capable of comparing two `Person` objects by the alphabetical order of their names and addresses:

```
class Person: public Sortable
{
    public:
        // constructors, destructor, and stuff
```

```
        Person();
        Person(char const *nm, char const *add, char const *ph);
        Person(Person const &other);
        Person const &operator=(Person const &other);
        ~Person();

        // interface
        char const *getname() const;
        char const *getaddress() const;
        char const *getphone() const;
        void setname(char const *nm);
        void setaddress(char const *add);
        void setphone(char const *ph);

        // requirements enforced by Sortable
        int compare(Sortable const &other) const;

    private:
        // data members
        char *name, *address, *phone;
};

int Person::compare(Sortable const &o)
{
    Person
        const &other = (Person const &)o;
    register int
        cmp;

    return
    (
        // first try: if names unequal, we're done
        (cmp = strcmp(name, other.name)) ?
            cmp
        :
            // second try: compare by addresses
            strcmp(address, other.address)
    );
}
```

Note in the implementation of `Person::compare()` that the argument of the function is not a reference to a `Person` but a reference to a `Sortable`. Remember that **C++** allows function overloading: a function `compare(Person const &other)` would be an entirely different function from the one required by the protocol of `Sortable`. In the implementation of the function we therefore cast the `Sortable&` argument to a `Person&` argument.

## 16.3 Comparing only Persons

Sometimes it may be useful to know in the concrete implementation of a pure virtual function what the `other` object is. E.g., the function `Person::compare()` should make the comparison `only` if the `other` object is a `Person` too: imagine what the expression

```
strcmp(name, other.name)
```

would do when the `other` object were in fact not a `Person` and hence did not have a `char *name` datamember.

We therefore present here an improved version of the protocol of the class `Sortable`. This class is expanded to require that each derived class implements a function `int getsignature()`:

```
class Sortable
{
    ...
    virtual int getsignature() const = 0;
    ...
};
```

The concrete function `Person::compare()` can now compare names and addresses only if the signatures of the current and other object match:

```
int Person::compare(Sortable const &o)
{
    register int
        cmp;

    // first, check signatures
    if ((cmp = getsignature() - o.getsignature()))
        return (cmp);

    Person
        const &other = (Person const &)o;

    return
    (
        // next try: if names unequal, we're done
        (cmp = strcmp(name, other.name)) ?
            cmp
        :
            // last try: compare by addresses
            strcmp(address, other.address)
    );
}
```

The crux of the matter is of course the function `getsignature()`. This function should return a unique `int` value for its particular class. An elegant implementation is the following:

```
class Person: public Sortable
{
    ...
    // getsignature() now required too
    int getsignature() const;
}

int Person::getsignature() const
{
    static int              // Person's own tag, I'm quite sure
        tag;                // that no other class can access it

    return ((int) &tag);    // Hence, &tag is unique for Person
}
```

For the reader who's puzzled by our 'elegant solution': the `static int tag` defined in the `Person::getsignature()` function is just one variable, no matter how many `Person` objects exist. Furthermore, it's created compile-time as a global variable, since it's static. Hence, there's only one variable `tag` for the `Person` class. Its *address*, therefore, is uniquely connected to the `Person` class. This `address` is cast to an `int` which thus becomes the (unique) signature of `Person` objects.

## 16.4 Virtual destructors

When the operator `delete` releases memory which is occupied by a dynamically allocated object, a corresponding destructor is called to ensure that internally used memory of the object can also be released. Now consider the following code fragment, in which the two classes from the previous sections are used:

```
Sortable
    *sp;
Person
    *pp = new Person("Frank", "frank@icce.rug.nl", "363 3688");

sp = pp;            // sp now points to a Person
...
delete sp;          // object destroyed
```

In this example an object of a derived class (`Person`) is destroyed using a base class pointer (`Sortable *`). For a 'standard' class definition this will mean that the destructor of `Sortable` is called, instead of the destructor of `Person`.

**C++** however allows a destructor to be virtual. By preceding the declaration of a destructor with the keyword `virtual` we can ensure that the right destructor is activated even when called via a base class pointer. The definition of the class `Sortable` would therefore become:

```
class Sortable
{
    public:
        virtual ~Sortable();
        virtual int compare(Sortable const &other) const = 0;
        ...
};
```

Should the virtual destructor of the base class be a *pure* virtual function or not? In general, the answer to this question would be no: for a class such as `Sortable` the definition should not *force* derived classes to define a destructor. In contrast, `compare()` is a pure virtual function: in this case the base class defines a protocol which must be adhered to.

By defining the destructor of the base class as `virtual`, but not as purely so, the base class offers the possibility of redefinition of the destructor in any derived classes. The base class doesn't enforce the choice.

The conclusion is therefore that the base class must define a destructor function, which is used in the case that derived classes do *not* define their own destructors. Such a destructor could be an empty function:

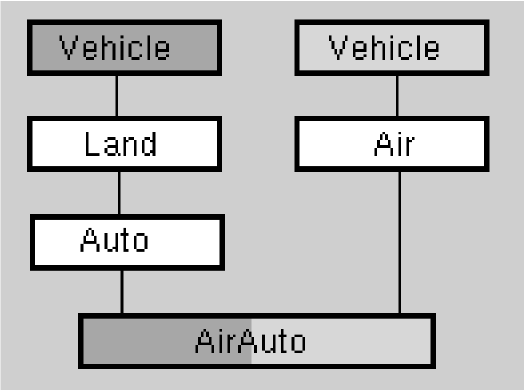
```
Sortable::~Sortable()
{
}
```

## 16.5 Virtual functions in multiple inheritance

As was previously mentioned in chapter 15 it is possible to derive a class from several base classes at once. Such a derived class inherits the properties of all its base classes. Of course, the base classes themselves may be derived from classes yet higher in the hierarchy.

A slight difficulty in multiple inheritance may arise when more than one 'path' leads from the derived class to the base class. This is illustrated in the code fragment below: a class `Derived` is doubly derived from a class `Base`:

```
class Base
{
    public:
        void setfield(int val)
            { field = val; }
        int getfield() const
            { return (field); }
    private:
```




\

Figure 16.2: Duplication of a base class in multiple derivation.

```
        int field;
};

class Derived: public Base, public Base
{
};
```

Due to the double derivation, the functionality of `Base` now occurs twice in `Derived`. This leads to ambiguity: when the function `setfield()` is called for a `Derived` object, *which* function should that be, since there are two? In such a duplicate derivation, many C++ compilers will fail to generate code and (correctly) identify the error.

The above code clearly duplicates its base class in the derivation. Such a duplication can be easily avoided here. But duplication of a base class can also occur via nested inheritance, where an object is derived from, say, an `Auto` and from an `Air` (see the vehicle classification system, chapter 15.1). 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 `getweight()` functions.

### 16.5.1 Ambiguity in multiple inheritance

Let's investigate closer why an `AirAuto` introduces ambiguity, when derived from `Auto` and `Air`.

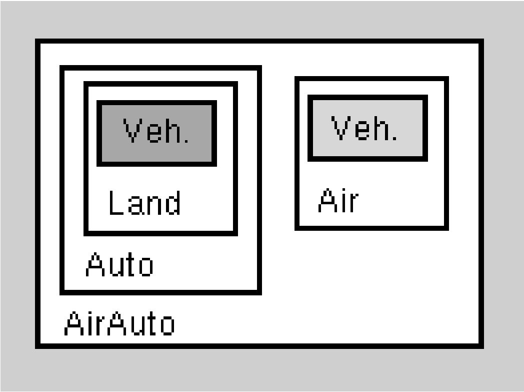
- An `AirAuto` is an `Auto`, hence a `Land`, and hence a `Vehicle`.
- However, an `AirAuto` is also an `Air`, and hence a `Vehicle`.

The duplication of `Vehicle` data is further illustrated in figure 16.2.

The internal organization of an `AirAuto` is shown in figure 16.3

The **C++** compiler will detect the ambiguity in an `AirAuto` object, and will therefore fail to produce code for a statement like:

[2]such as the one in James Bond vs. the Man with the Golden Gun...




\

Figure 16.3: Internal organization of an `AirAuto` object.

```
AirAuto
    cool;

printf("%d\n", cool.getweight());
```

The question of which member function `getweight()` should be called, cannot be resolved by the compiler. The programmer has two possibilities to resolve the ambiguity explicitly:

- First, the function call where the ambiguity occurs can be modified. This is done with the scope resolution operator:

  ```
  // let's hope that the weight is kept in the Auto
  // part of the object..
  printf("%d\n", cool.Auto::getweight());
  ```

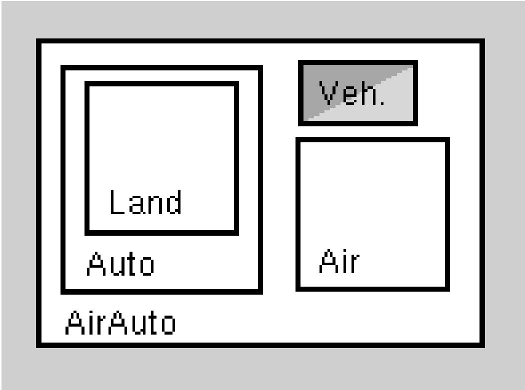
  Note the place of the scope operator and the class name: before the name of the member function itself.

- Second, a dedicated function `getweight()` could be created for the class `AirAuto`:

  ```
  int AirAuto::getweight() const
  {
      return(Auto::getweight());
  }
  ```

The second possibility from the two above is preferable, since it relieves the programmer who uses the class `AirAuto` of special precautions.

However, besides these explicit solutions, there is a more elegant one. This will be discussed in the next section.




\

Figure 16.4: Internal organization of an `AirAuto` object when the base classes are virtual.

### 16.5.2 Virtual base classes

As is illustrated in figure 16.3, more than one object of the type `Vehicle` is present in one `AirAuto`. The result is not only an ambiguity in the functions which access the `weight` data, but also the presence of two `weight` fields. This is somewhat redundant, since we can assume that an `AirAuto` has just one weight.

We can achieve that only one *Vehicle* will be contained in an `AirAuto`. This is done by ensuring that the base class which is multiply present in a derived class, is defined as a *virtual base class*. The behavior of virtual base classes is the following: when a base class `B` is a virtual base class of a derived class `D`, then `B` may be present in `D` but this is not necessarily so. The compiler will leave out the inclusion of the members of `B` when these are already present in `D`.

For the class `AirAuto` this means that the derivation of `Land` and `Air` is changed:

```
class Land: virtual public Vehicle
{
    ...
};

class Air: virtual public Vehicle
{
    ...
};
```

The virtual derivation ensures that via the `Land` route, a `Vehicle` is only added to a class when not yet present. The same holds true for the `Air` route. This means that we can no longer say by which route a `Vehicle` becomes a part of an `AirAuto`; we only can say that there is one `Vehicle` object embedded.

The internal organization of an `AirAuto` after virtual derivation is shown in figure 16.4.

With respect to virtual derivation we note:

- 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.
- In the above example it would suffice to define either `Land` or `Air` with virtual derivation. That also would have the effect that one definition of a `Vehicle` in an `AirAuto` would be dropped. Defining both `Land` and `Air` as virtually derived is however by no means erroneous.
- The fact that the `Vehicle` in an `AirAuto` is no longer 'embedded' in `Auto` or `Air` has a consequence for the chain of construction. The constructor of an `AirAuto` will directly call the constructor of a `Vehicle`; this constructor will not be called from the constructors of `Auto` or `Air`.

Summarizing, virtual derivation has the consequence that ambiguity in the calling of member functions of a base class is avoided. Furthermore, duplication of data members is avoided.

### 16.5.3 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 15.4:

```
class Truck: public Auto
{
    public:
        // constructors
        Truck();
        Truck(int engine_wt, int sp, char const *nm,
               int trailer_wt);

        // interface: to set two weight fields
        void setweight(int engine_wt, int trailer_wt);
        // and to return combined weight
        int getweight() const;

    private:
        // data
        int trailer_weight;
};

// example of constructor
Truck::Truck(int engine_wt, int sp, char const *nm,
              int trailer_wt)
:
    Auto(engine_wt, sp, nm)
{
    trailer_weight = trailer_wt;
}

// example of interface function
```

```
int Truck::getweight() const
{
    return
    (                               // sum of:
        Auto::getweight() +         //   engine part plus
        trailer_wt                  //   the trailer
    );
}
```

This definition shows how a `Truck` object is constructed to hold two weight fields: one via its derivation from `Auto` and one via its own `int trailer_weight` data member. Such a definition is of course valid, but could be rewritten. We could let a `Truck` be derived 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 item 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 resolves 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 resolved in the class `Truck`:

```
class TrailerVeh: public Vehicle
{
    public:
        TrailerVeh(int wt);
};

TrailerVeh::TrailerVeh(int wt)
:
    Vehicle(wt)
{
}

class Truck: public Auto, public TrailerVeh
{
    public:
        // constructors
        Truck();
        Truck(int engine_wt, int sp, char const *nm,
               int trailer_wt);

        // interface: to set two weight fields
        void setweight(int engine_wt, int trailer_wt);
        // and to return combined weight
```

```
        int getweight() const;
};

// example of constructor
Truck::Truck(int engine_wt, int sp, char const *nm,
              int trailer_wt)
:
    Auto(engine_wt, sp, nm),
    TrailerVeh(trailer_wt)
{
}

// example of interface function
int Truck::getweight() const
{
    return
        (                              // sum of:
            Auto::getweight() +        //   engine part plus
            TrailerVeh::getweight()    //   the trailer
        );
}
```

## 16.6 Run-Time Type identification

**C++** offers two ways to retrieve the type of objects and expressions while the program is run. The possibilities of **C++**'s *run-time type identification* are somewhat limited compared to languages like **JAVA**. Normally, **C++** uses static type checking and type identification. Static type checking and determination is safer and 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 pointer or reference to a base class to a pointer or reference to a derived class.
- The `typeid` operator returns the actual type of an expression.

For all practical purposes, these operators work on class type objects, where the classes contain one or more virtual functions.

### 16.6.1 The dynamic_cast operator

The `dynamic_cast` operator is used to convert a (base) class pointer or reference to a (base) class object to, respectively, a derived class pointer or derived class reference.

The dynamic cast is performed run-time. A prerequisiste for the proper functioning of the dynamic cast operator is the existence of at least one virtual function in the base class.

In the following example a pointer to the class `Derived` is obtained from the `Base` class pointer `bp`:

```
class Base
{
    public:
        virtual ~Base();
};
class Derived: public Base
{
    public:
        char const *toString()
        {
            return ("Derived object");
        }
};
int main()
{
    Base
        *bp;
    Derived
        *dp,
        d;

    bp = &d;

    if ((dp = dynamic_cast<Derived *>(bp)))
        cout << dp->toString() << endl;
    else
        cout << "dynamic cast conversion failed\n";

    return (0);
}
```

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 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:

```
class Base
{
    public:
        virtual ~Base();
};
class Derived: public Base
{
    public:
        char const *toString()
        {
            return ("Derived object");
        }
```

```
};
class SecondDerived: public Base
{
    public:
        char const *hello()
        {
            return ("hello from a SecondDerived object");
        }
};
int main()
{
    Base
        *bp;
    Derived
        *dp,
        d;
    SecondDerived
        *sdp;

    bp = &d;

    if ((dp = dynamic_cast<Derived *>(bp)))
        cout << dp->toString() << endl;
    else if ((sdp = dynamic_cast<SecondDerived *>(bp)))
        cout << dp->hello() << 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, assuming the availability of the above-mentioned classes `Base, Derived`, and `SecondDerived`:

```
void process(Base &b)
{
    try
    {
        cout << dynamic_cast<Derived &>(b).toString() << endl;
        return;
    }
    catch (std::bad_cast)
    {}

    try
    {
        cout << dynamic_cast<SecondDerived &>(b).hello() << endl;
        return;
    }
    catch (std::bad_cast)
    {}
}
```

```
int main()
{
    Derived
        d;

    process(d);
    return (0);
}
```

In this example the value `std::bad_cast` is introduced. The `std::bad_cast` is thrown as an exception if the dynamic cast of a reference to a base class object fails.

The dynamic cast operator may be a handy tool when an existing base class cannot or should not be modified (e.g., when the sources are not available), and a derived class may be modified instead. Code receiving a base class pointer or reference may then perform a dynamic cast to the derived class to be able to use the derived class' functionality.

Casts from a base class reference or pointer to a derived class reference or pointer are called *downcasts*.

### 16.6.2 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, the header file `typeinfo` must be included:

```
#include <typeinfo>
```

Actually, the `typeid` operator returns an object of type `type_info`, which may, e.g., be compared to other `type_info` objects.

The class `type_info` may be implemented differently by different implementations, but at the very least it has the following interface:

```
class type_info
{
    public:
        virtual ~type_info();
        int operator==(const type_info &other) const;
        int operator!=(const type_info &other) const;
        char const *name() const;
    private:
        type_info(type_info const &other);
        type_info &operator=(type_info const &other);
};
```

Note that this class has a private copy constructor and overloaded assignment operator. This prevents the normal construction or assignment of a `type_info` object. `Type_info` objects are constructed and returned by the `typeid` operator. Implementations, however, may choose to extend or elaborate upon the `type_info` class and provide, e.g., lists of functions that can be called in 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 >= 1 virtual functions
class Derived: public Base;

Derived
    d;
Base
    &br = d;

cout << typeid(br).name() << endl;
```

In this example the `typeid` operator is given a base class reference. It will print the text `Derived`, being the class name of the class `br` actually refers to. If `Base` does not contain virtual functions, the text `Base` would have been printed.

The `typeid` operator can be used to determine the name of any type of expression, not just of class type objects. For example:

```
cout << typeid(12).name() << endl;     // prints:  int
cout << typeid(12.23).name() << endl;  // prints:  double
```

Note, however, that the above example is suggestive at most of the type that is printed. It *may* be `int` and `double`, but this is not necessarily the case. If portability is required, make sure no tests against static, built-in strings are required. Check out what your compiler produces in case of doubt.

In situations where the `typeid` operator is applied to determine the type of a derived class, it is important to realize that a base class *reference* is used as the argument of the `typeid` operator. Consider the following example:

```
class Base;     // contains at least one virtual function
class Derived: public Base;

Base
    *bp = new Derived;      // base class pointer to derived object

if (typeid(bp) == typeid(Derived *))    // 1: false
    ...
if (typeid(bp) == typeid(Base *))       // 2: true
```

```
    ...
if (typeid(bp) == typeid(Derived))      // 3: false
    ...
if (typeid(bp) == typeid(Base))         // 4: false
    ...
```

Here, `(1)` returns `false` as a `Base` ∗ is not a `Derived` ∗. `(2)` returns `true`, as the two pointer types are the same, `(3)` and `(4)` return `false` as pointers to objects are not the objects themselves.

On the other hand, if ∗`bp` is used in the above expressions, then `(1)` and `(2)` return `false` as an object (or reference to an object) is not a pointer to an object, whereas with

```
if (typeid(*bp) == typeid(Derived))      // 3: true
    ...
if (typeid(*bp) == typeid(Base))         // 4: false
    ...
```

we see that `(3)` now returns `true`: ∗`bp` actually refers to a `Derived` class object, and `typeid(`∗`bp)` will return `typeid(Derived)`.

A similar result is obtained if a base class reference is used:

```
Base
    &br = *bp;

if (typeid(br) == typeid(Derived))      // 3: true
    ...
if (typeid(br) == typeid(Base))         // 4: false
    ...
```

# Chapter 17

# Templates

The **C++** language support a mechanism which allows programmers to define completely general functions or classes, based on hypothetical arguments or other entities. Code in which this mechanism has been used is found in de chapter on abstract containers.

These general functions or classes become concrete code once their definitions are applied to real entities. The general definitions of functions or classes are called *templates*, the concrete implementations *instantiations*.

In this chapter we will examine template functions and template classes.

## 17.1 Template functions

Template functions are used in cases where a single implementation of a function is not practical due to the different types that are distinguished in **C++**. If a function is defined as

```
fun(int *array)
```

then this function will likely run into problems if it is passed the address of an array of `double` values. The function will normally have to be duplicated for parameters of different types. For example, a function computing the sum of the elements of an array for an array of `ints` is:

```
int sumVector(int *array, unsigned n)
{
    int
        sum(0);
    for (int idx = 0; idx < n; ++idx)
        sum += array[idx];
    return (sum);
}
```

The function must be overloaded for arrays of doubles:

```
double sumVector(double *array, unsigned n)
{
    double
        sum(0);
    for (int idx = 0; idx < n; ++idx)
        sum += array[idx];
    return (sum);
}
```

In a local program development situation this hardly ever happens, since only one or two `sumVector()` implementations will be required. But the strongly typed nature of **C++** stands in the way of creating a truly general function, that can be used for any type of array.

In cases like these, *template functions* are used to create the truly general function. The template function can be considered a general *recipe* for constructing a function that can be used with the general array. In the coming sections we'll discuss the construction of template functions. First, the construction of a template function is discussed. Then the instantiation is covered. With template functions the argument deduction deserves special attention, which is given in section 17.1.3.

### 17.1.1 Template function definitions

The definition of a template function is very similar to the definition of a normal function, except for the fact that the parameters, the types that are used in the function, and the function's return value may be specified in a completely general way. The function `sumVector()` in the previous section can as follows be rewritten as a template function:

```
template <class T>
T sumVector(T *array, unsigned n)
{
    T
        sum(0);
    for (int idx = 0; idx < n; ++idx)
        sum += array[idx];
    return (sum);
}
```

Note the correspondence with the formerly defined `sumVector()` functions. In fact, if a `typedef int T` had been specified, the template function, except for the initial `template` line, *would* be the first `sumVector()` function of the previous section. So, the essence of the template function is found in the first line. From the above example:

`template <class T>`

This line starts out the definition or declaration of a template function. It is followed by the *template parameter list*, which is a comma-separated non-empty list of so-called *template type* or *template non-type parameters*, surrounded by angular brackets < and >. In the template function `sumVector()` the only template parameter is `T`, which is a template type parameter. ttT) is the formal type that is used in the template function definition to represent the actual type that will be specified when the template function is instantiated. This type is used in the parameter list of the function, it is used to define the type of a local variable of the function, and it is used to define the return type of the function.

Normal scope rules and identifier rules apply to template definitions and declarations: the type `T` is a *formal name*, it could have been named `Type`. The formal typename that is used overrules, within the scope of the template definition or declaration, any previously defined identifiers by that name.

A template *non-type* parameter represents a constant expression, which must be known by the time the template is instantiated, and which is specified in terms of existing types, such as an `unsigned`.

An alternative definition for the above template function, using a template non-type parameter is:

```
template <class T, unsigned size>
T sumVector(const T (&array)[size])
{
    T
        sum(0);
    for (int idx = 0; idx < size; ++idx)
        sum += array[idx];
    return (sum);
}
```

Template function definitions may have multiple type and non-type parameters. Each parameter name must be unique. For example, the following template declaration declares a template function for a function `outerProduct()`, returning a pointer to vectors of `size2 T2` elements, and expecting two vectors of, respectively, `size1` and `size2` elements:

```
template
<
    class T1,
    class T2,
    unsigned size1,
    unsigned size2
>
    T1
    (
        *outerProduct
        (
            T2 const (&v1)[size1],
            T2 const (&v2)[size2]
        )
    )[size2];
```

Note that the return type `T1` of the returned vectors is intentionally specified different from `T2`. This allows us to specify, e.g., return type `double` for the returned outer product, while the vectors passed to `outerProduct` are of type `int`. Instead of using the keyword `class`, the keyword `typename` can be used in template type parameter lists. However, the keyword `typename` is *required* in certain situations that may occur when the template function is defined. For example, assume we define the following template function:

```
template <class T>
void function()
{
    unsigned
        p;
    ...
    {
        T::member
            *p;
        ...
    }
}
```

Although the layout of the above function suggests that `p` is defined as a pointer to the type `member`, that must have been declared in the class that is specified when the function is instantiated, it actually is interpreted by the compiler as a multiplication of `T::member` and `p`.

The compiler does so, because it cannot know from the template definition whether `member` is a *typename*, defined in the class `T`, or a *member* of the class `T`. It takes the latter and, consequently, interprets the $*$ as a multiplication operator.

What if this interpretation was not intended? In that case the `typename` keyword must be used. In the following template definition the $*$ indicates a pointer definition to a `T::member` type.

```
template <class T>
void function()
{
    unsigned
        p;
    ...
    {
        typename T::member
            *p;
        ...
    }
}
```

### The keyword 'typename'

As illustrated in section 17.1.1 The keyword `typename` can be used to disambiguate members and typenames in cases where the template type parameter represents a class type. It can also be used instead of the `class` keyword indicating a template type. So, instead of

```
template <class T>
void function(T type)
{
    ...
}
```

the function can be defined as:

```
template <typename T>
void function(T type)
{
    ...
}
```

### 17.1.2 Instantiations of template functions

Consider the first template function definition in section 17.1.1. This definition is a mere recipe for constructing a particular function. The function is actually constructed once it is used, or its address is taken. Its type is implicitly defined by the nature of its parameters.

For example, in the following code assumes that the function `sumVector` has been defined in the header file `sumvector.h`. In the function `main()` the function `sumVector()` is called once for the `int` array `x`, once for the `double` array `y`, and once the address is taken of a `sumVector()` function. By taking the address of a `sumVector` function the type of the argument is defined by the type of the pointer variable, in this case a pointer to a function processing a array of `unsigned long` values. Since such a function wasn't available yet (we had functions for `ints` and `doubles`, it is constructed once its address is required. Here is the function `main()`:

```
#include "sumvector.h"

int main()
{
    int
        x[] = {1, 2};
    double
        y[] = {1.1, 2.2};

    cout << sumVector(x, 2) << endl      // first instantiation
        << sumVector(y, 2) << endl;      // second instantiation

    unsigned long                         // third instantiation
        (*pf)(unsigned long *, unsigned) = sumVector;

    return (0);
```

```
}
```

While in the above example the functions `sumVector()` could be instantiated, this is not always possible. Consider the following code:

```
#include "template.h"

unsigned fun(unsigned (*f)(unsigned *p, unsigned n));
double fun(double (*f)(double *p, unsigned n));

int main()
{
    cout << fun(sumVector) << endl;
    return (0);
}
```

In the above example the function `fun()` is called in the function `main()`. Although it appears that the address of the function `sumVector()` is passed over to the function `fun()`, there is a slight problem: there are two overloaded versions of the function `fun()`, and *both* can be given the address of a function `sumVector()`. The first function `fun()` expects an `unsigned` $*$, the second one a `double` $*$. Which instantiation must be used for `sumVector()` in the `fun(sumVector)` expression? This is an ambiguity, which balks the compiler. The compiler complains with a message like

```
In function ‘int main()’:
call of overloaded ‘fun ({unknown type})’ is ambiguous
candidates are: fun(unsigned int (*)(unsigned int *, unsigned int))
                fun(double (*)(double *, unsigned int))
```

Situations like this should of course be avoided. Template functions can only be instantiated if this can be done unambiguously. It is, however, possible to disambiguate the situation using a cast. In the following code fragment the (proper) `double` $*$ implementation is forced by means of a `static_cast`:

```
#include "template.h"

unsigned fun(unsigned (*f)(unsigned *p, unsigned n));
double fun(double (*f)(double *p, unsigned n));

int main()
{
    cout << fun(static_cast<double (*)(double *, unsigned)>(sumVector))
        << endl;
    return (0);
}
```

But casts should be avoided, where possible. Fortunately the cast can be avoided in this kind of situation, as described in section 17.1.4.

If the same template function definition was included in different source files, which are then compiled to different object files which are thereupon linked together, there will, per type of template function, be only one instantiation of the template function in the final program.

This is illustrated by the following example, in which the address of a function `sumVector()` for `int` arrays is written to `cout`. The first part defines a function `fun()` in which the address of a `sumVector()` function is written to `cout`. The second part defines a function `main()`, defined in a different sourcefile, in which the address of a similar `sumVector()` function is written to `cout`, and in which `fun()` is called:

```
// This is source file 1: fun.cc
#include "template.h"
void fun()
{
    cout << static_cast<void *>
            (
                static_cast<int (*)(int *, unsigned)>
                (sumVector)
            )
        << endl;
}

// This is source file 2: main.cc
#include "template.h"

void fun();

int main()
{
    fun();

    cout << static_cast<void *>
            (
                static_cast<int (*)(int *, unsigned)>
                (sumVector)
            )
        << endl;

    return (0);
}
```

After compiling and linking the above two source files, the resulting program produces output like:

```
0x8048760
0x8048760
```

the addresses of the two functions are the same, so each function eventually uses the same implementation of the template function.

Knowing this, it is also understandable that it is possible to *declare* a template function, if it is *known* that the required instantiation is available in another sourcefile. E.g., the function `fun()` in the above example could be defined as follows:

```
template<class T>
T sumVector(T *tp, unsigned n);

void fun()
{
    cout << static_cast<void *>
            (
                static_cast<int (*)(int *, unsigned)>
                (sumVector)
            )
        << endl;
}
```

To make this work, one must of course be certain that the instantiation is available elsewhere. The *advantage* of this approach is that the compiler doesn't have to instantiate a template function, which speeds up the compilation of the function `fun()`, the *disadvantage* is that we have to do the bookkeeping ourselves: is the template function used somewhere else or not?

A third approach, is to *declare* template functions in header files, keeping the definition in a template source file. In the template source file the functions are instantiated by pointers to the appropriate functions. For example, define `sumvector.cc` as follows:

```
template<class T>
T sumVector(T *tp, unsigned n)
{
    return (*tp);
}

static void
    *p1 = static_cast<int (*)(int *, unsigned)>(sumVector);
```

and declare the `sumVector` template function in all sourcefiles using `sumVector`. This way the compiler keeps track of which `sumVector()` functions are required, linking them from the `sumvector.o` object when necessary. Of course, they must be available there. But if they aren't then they can be defined simply by providing another pointer defnition, followed by a recompilation of `sumvector.cc`. The advantage here is gain in compilation time (and maybe a clear overview of what template functions are actually instantiated), as well as *data hiding*: the implementation of the template function is not required by the users of the implementation, and can therefore be hidden from them. The disadvantage is the definition of a bunch of `static void` $*$ variables: they are used as `rvalues` for the addresses of instantiated template functions. Another disadvantage is that the template definition is not available for other situations. If some program would benefit from a sumVector() instantiation for a type that is not available in sumvector.cc, the template itself or the sumvector.cc sourcefile would be required (since we strongly agree with the principles of the free software foundation, the latter disadvantage is actually more of an *advantage* in our opinion :-).

Finally, as the structure of the void * definitions is always the same, a macro definition might come in handy here. E.g., the sumvector.cc source file in which three sumVector() functions are instantiated could be written as follows:

```
template<class T>
T sumVector(T *tp, unsigned n)
{
    return (*tp);
}
                    // NOTE: the next line ends at the backslash
#define instantiate(type)   \
    static_cast<type (*)(type *, unsigned)>(sumVector)

static void
    *p[] =
    {
        instantiate(int),
        instantiate(double),
        instantiate(unsigned)
    };

#undef instantiate
```

This model can be used over and over again: the instantiate() macro is never defined outside of the sourcefile itself, while instantiations can be generated on the fly by new instantiate() macro calls.

### 17.1.3 Argument deduction

The compiler determines what type of template function is needed by examining the types and values of the arguments of template functions. This process is called *template argument deduction.* With template argument deduction, the type of the return value of the template function is not considered.

For example, consider once again the function

T sumVector(const T (&array)[size])

given in section 17.1.1:

```
    template <class T, unsigned size>
    T sumVector(const T (&array)[size])
    {
```

```
        T
            sum(0);
        for (int idx = 0; idx < size; ++idx)
            sum += array[idx];
        return (sum);
    }
```

In this function the template non-type parameter `size` is determined from the size of the array that is used with the call. Since the size of an array is known to the compiler, the compiler can determine the `size` parameter by looking up the size of the array that is used as argument to the function `sumVector()`. If the size is not known, e.g., when a pointer to an array element is passed to the function, the compilation will not succeed. Therefore, in the following example, the first call of the function `sumVector()` will succeed, as `iArray` is an array; the second one will fail, as `iPtr` is a pointer, pointing to an array of (in principle) unknown size:

```
#include "sumvector.t"  // define the template function

int main()
{
    int
        iArray[] = {1, 2, 3},
        *iPtr = iArray;

    sumVector(iArray);  // succeeds: size of iArray is known
    sumVector(iPtr);    // fails: size of array pointed to by
                        // iPtr is unknown
    return (0);
}
```

It is not necessary for a template function's argument to match exactly the type of the template function's corresponding parameter. Three kinds of conversions are allowed here:

- *lvalue transformations*
- *qualification conversions*
- *conversion to a base class instantiated from a class template*

These three conversions are now discussed and illustrated.

### Lvalue transformations

There are three types of *lvalue transformations*:

- *lvalue-to-rvalue conversions*
- *array-to-pointer conversions*
- *function-to-pointer conversions*

**lvalue-to-rvalue conversions.** Simply stated, an *lvalue* is an expression that may be used to the left of an assignment operator. It is an object whose address may be determined, and which contains a value. In contrast, an *rvalue* is an expression that may be used to the right of an assignment operator: it represents a value that does not have an address and that cannot be modified.

In a statement like

```
x = y;
```

(in which `x` and `y` are variables of comparable types), the value of `y` is determined. Then this value is assigned to `x`. Determining the value of `y` is called an *lvalue-to-rvalue conversion*. An lvalue-to-rvalue conversion takes place in situations where the value of an `lvalue` expression is required. This also happens when a variable is used as argument to a function having a *value parameter*.

**array-to-pointer conversions.** An *array-to-pointer conversion* occurs when the name of an array is assign to a pointervariable. This if frequently seen with functions using parameters that are pointer variables. When calling such functions, an array is often specified as argument to the function. The address of the array is then assigned to the pointer-parameter. This is called an *array-to-pointer conversion*.

**function-to-pointer conversions.** This conversion is most often seen with functions defining a parameter which is a pointer to a function. When calling such a function the name of a function may be specified for the parameter which is a pointer to a function. The address of the function is then assigned to the pointer-parameter. This is called a *function-to-pointer conversion*.

In the first sumVector() function (section 17.1.1) the first parameter is defined as a `T *`. Here an array-to-pointer conversion is allowed, as it is an lvalue transformation, which is one of the three allowed conversions. Therefore, the name of an array may be passed to this function as its first argument.

### Qualification conversions

A *qualification conversion* adds `const` or `volatile` qualifications to pointers. Assume the function sumVector() in section 17.1.1 was defined as follows:

```
template <class T>
T sumVector(T const *array, unsigned n)
{
    T
        sum(0);
    for (int idx = 0; idx < n; ++idx)
        sum += array[idx];
    return (sum);
}
```

In the above definition, a plain array or pointer to some type can be used in combination with this function `sumVector()`. E.g., an argument `iArray` could be defined as `int iArray[5]`. However, no damage is inflicted on the elements of `iArray` by the function `sumVector()`: it explicitly states so, by defining `array` as a `T const *`. Qualification conversions are therefore allowed in the process of template argument deduction.

### Conversion to a base class

In section 17.2 template classes are formally introduced. However, they were already *used* earlier: abstract containers (covered in chapter 8) are actually defined as template classes. Like 'normal' classes, template classes can participate in the construction of class hierarchies. In section 17.2.7 it is shown how a template class can be derived from another template class.

Assume that the template class `Pipe` is derived from the class `queue`. Furthermore, assume our function `sumVector()` was written to return the sum of the elements of a `queue`:

```
template <class T>
T sumVector(queue<T> &queue)
{
    T
        sum(0);

    while (!queue.empty())
    {
        sum += gueue.front();
        queue.pop();
    }
    return (sum);
}
```

All kinds of `queue` objects can be passed to the above function. However, it is also possible to pass `Pipe` objects to the function `sumVector()`: By instantiating the `Pipe` object, its base class, which is the template class `queue`, is also instantiated. Now:

- `Pipe<xxx>` has `queue<xxx>` as its base class, and
- `queue<xxx>` is a possible first argument of the above template function `sumVector()`, and
- a function argument which is of a derived class type may be used with a base class parameter of a template function.

Consequently, the definition '`Pipe<int> pi;`' implies the instantiation of the base class `queue<int>`, which is an allowed type for the first parameter of `sumVector()`. Therefore, `pi` may be passed as argument to `sumVector()`.

This conversion is called a *conversion to a base class instantiated from a class template.* In the above example, the *class template* is `Pipe`, the base class is `queue`.

### Summary: the template argument deduction algorithm

The following algorithm is used with template argument deduction when a template function is called with one or more arguments:

- In turn, the template parameters are identified in the parameters of the called function.
- For each template parameter, the template's type is deduced from the template function's argument (e.g., `int` if the argument is a `Pipe<int> object`).
- The three allowed conversions (see section 17.1.3) for template arguments are applied where necessary.
- If the same template parameter is used with multiple function parameters, the template types of the arguments must be the same. E.g., with template function

  ```
  twoVectors(vector<Type> &v1, vector<Type> &v2)
  ```

  the arguments used with `twoVectors()` must have equal types. E.g.,

  ```
  vector<int>
      v1,
      v2;
  ...
  twoVectors(v1, v2);
  ```

### 17.1.4 Explicit arguments

Consider once again the function main() is section 17.1.2. Here the function `sumVector()` was called as follows:

```
#include "sumvector.h"

int main()
{
    int
        x[] = {1, 2};
    double
        y[] = {1.1, 2.2};

    cout << sumVector(x, 2) << endl
        << sumVector(y, 2) << endl;
    ...
}
```

In both cases the final argument of the function is of type `int`, but in the template's definition, the second parameter is an unsigned. The conversion `unsigned -> int` is not one of the allowed conversions *lvalue transformations, qualification conversions* or `conversion to a base class`. Why doesn't the compiler complain in this case? In cases where the type of the argument is fixed, *standard type conversions* are allowed, and they are applied automatically by the compiler. The types of arguments may also be made explicit by providing casts. In those cases there is no need for the compiler to deduce the types of the arguments.

In section 17.1.2, a cast was used to disambiguate. Rather than using a `static_cast`, the type of the required function can be made explicit using another syntax: the function name may be followed by the types of the arguments, surrounded by pointed brackets. Here is the example of section 17.1.2 using explicit template argument types:

```
#include "template.h"

unsigned fun(unsigned (*f)(unsigned *p, unsigned n));
double fun(double (*f)(double *p, unsigned n));

int main()
{
    cout << fun(sumVector<double, unsigned>)
        << endl;
    return (0);
}
```

The explicit argument type list should follow the types mentioned in the `template<...>` line preceding the template's function definition. The type `class T` in the template line of the function `sumVector()` is made explicit as type `double`, and not as, e.g., a type `double *`, which was used in the `static_cast` in the example of section 17.1.2.

Explicit template arguments may be partially specified. Like the specification of arguments of functions for which default arguments are defined, trailing template arguments may be omitted from the list of explicit template argument types. When they are omitted, the types mentioned in the `template<>` line preceding the template's function definition are used. So, in the above example the explicit argument type `unsigned` may be omitted safely, as the type of the second template's argument is already known from the argument type list. The function `main()` can therefore also be written as:

```
int main()
{
    cout << fun(sumVector<double>)
        << endl;
    return (0);
}
```

Explicit template arguments can also be used to simplify the definition of the instantiate macro in section 17.1.2. Using an explicit template argument, the code gets so simple that the macro itself can be completely avoided. Here is the revised code of the example:

```
template<class T>
T sumVector(T *tp, unsigned n)
{
    return (*tp);
}
static void
```

```
*p[] =
{
    &sumVector<int>,
    &sumVector<double>,
    &sumVector<unsigned>
};
```

Note that the initial &-tokens indicating the addresses of the `sumVector()` functions are required when the addresses of the functions are assigned to pointer variables.

**Template explicit instantiation declarations**

The explicit instantiations that were defined in the previous section were all embedded in the array of void pointers `p[]`, which array was used to have a target for the addresses of the instantiated function.

This is, admittedly not too elegant, but it works well. However, it is also possible to *declare* a template providing explicit types of the template's arguments with the purpose of instantiating the corresponding template functions. An *explicit instantiation declaration* starts with the keyword `template`, to be followed by an explicit template function declaration. Although this is a declaration, it is considered by the compiler as a request to instantiate that particular variant of the function.

Using explicit instantiation declarations the final example of the previous section can be rewritten as follows:

```
template<class T>
T sumVector(T *tp, unsigned n)
{
    return (*tp);
}

template      int sumVector<int>(int *, unsigned);
template   double sumVector<double>(double *, unsigned);
template unsigned sumVector<unsigned>(unsigned *, unsigned);
```

As can be seen from this example, explicit instantiation declarations are mere function declarations, e.g.,

```
int sumVector(int *, unsigned);
```

embellished with the `template` keyword and an explicit template argument list, e.g., `<int>`.

### 17.1.5 Template explicit specialization

Although the function `sumVector()` we've seen in the previous sections is well suited for arrays of elements of the basic types (like `int`, `double`, etc.), the template implementation is of course not appropriate in cases where the `+=` operator is not defined or the `sum(0)` initialization makes no sense. In these cases an `template explicit specialization` may be provided,

The template's implementation of the `sumVector()` is not suited for variables of type `char *`, like the `argv` parameter of `main()`. If we want to be able to use `sumVector()` with variables of type `char *` as well, we can define the following special form of `sumVector()`:

```
#include <string>
#include <numeric>

template <> char *sumVector<char *>(char **argv, unsigned argc)
{
    string
        s = accumulate(argv, argv + argc, string());

    return (strcpy (new char[s.size() + 1], s.c_str()));
}
```

A template explicit specialization starts with the keyword `template`, followed by an empty set of pointed brackets. This is followed by the head of the function, which follows the same syntax as a template explicit instantiation declaration, albeit that the trailing `;` of the declaration is replaced by the actual function body of the specialization implementation.

The template explicit specialization is normally included in the same file as the standard implementation of the template function.

If the template explicit specialization is to be used in a different file than the file in which it is defined, it must be declared. Of course, being a template function, the definition of the template explicit specialization can also be included in every file in which it is used, but that will also slow down the compilation of those other files.

The declaration of a template explicit specialization obeys the standard syntax of a function declaration: the definition is replaced by a semicolon. Therefore, the declaration of the above template explicit specialization is

```
template <> char *sumVector<char *>(char **, unsigned);
```

Note the pair of pointed brackets following the `template` keyword. Were they omitted, the function would reduce to a template instantiation declaration: you would not notice it, except for the longer compilation time, as using a template instantiation declaration implies an extra instantiation (i.e., compilation) of the function.

In the *declaration* of the template explicit specialization the explicit specification of the template arguments (in the `< ... >` list following the name of the function) can be omitted if the types of the arguments can be deduced from the types of the arguments. With the above declaration this is the case. Therefore, the declaration can be simplified to:

```
template <> char *sumVector(char **, unsigned);
```

Comparably, the `template <>` part of the template explicit specialization may be omitted. The result is an ordinary function or ordinary function declaration. This is not an error: template functions and non-template functions may overload each other. Ordinary functions are less restrictive in the type conversions that are allowed for their arguments than template functions, which might be a reason for using an ordinary function. On the other hand, a template explicit specialization must obey the form of the general template function of which it is a specialization. If the template function head is

`T sumVector(T *tp, unsigned n)`

then the template explicit specialization cannot be

`template<> char *sumVector<char const *>(char const **, unsigned)`

as this results in different interpretations of the formal type `T` of the template: `char *` or `char const *`.

### 17.1.6 Overloading template functions

Template functions may be overloaded. The function `sumVector()` defined earlier (e.g. in section 17.1.1) may be overloaded to accept, e.g., variables of type `vector`:

```
#include <vector>
#include <numeric>

template <class T>
T sumVector(vector<T> &array)
{
    return (accumulate(array.begin(), array.end(), T(0)));
}
```

Such a template function can be used by passing it an argument of type `vector`, as in:

```
void fun(vector<int> &vi)
{
    cout << sumVector(vi) << endl;
}
```

Apart from defining overloaded versions, the overloaded versions can of course also be declared. E.g.,

```
template <class T>
T sumVector(vector<T> &array);
```

Using templates may result in ambiguities which overloading can't solve. Consider the following template function definition:

```
template<class T>
bool differentSigns(T v1, T v2)
{
    return
    (
        v1 < 0 && v2 >= 0
        ||
        v1 >= 0 && v2 < 0
    );
}
```

Passing `differentSigns()` an `int` and an `unsigned` is an error, as the two types are different, whereas the template definition calls for identical types. Overloading doesn't really help here: defining a template having the following prototype is ok with the `int` and `unsigned`, but now two instantiations are possible with *identical* types.

```
template<class T1, class T2>
bool differentSigns(T1 v1, T2 v2);
```

This situation can be disambiguated by using template explicit arguments, e.g., `differentSigns<int, int>(12, 30)`. But template explicit arguments could be used anyway with the second overloaded version of the function: the first definition is superfluous and can be omitted.

On the other hand, if one overloaded version can be interpreted as a *more specialized* variant of another version of a template function, then in principle the two variants of the template function could be used if the arguments are of the more specialized types. In this case, however, there is no ambiguity, as the compiler will use the more specialized variant if the arguments so suggest.

So, assume an overloaded version of `sumVector()` is defined having the following prototype and a snippet of code requiring the instantiation of `sumVector`:

```
template <class T>
T sumVector(T, unsigned);

extern int
    iArray[];

void fun()
{
    sumVector(iArray, 12);
}
```

The above example doesn't produce an ambiguity, even though the original sumVector() given in section 17.1.1 and the version declared here could both be used for the call. Why is there no ambiguity here?

In situations like this there is no ambiguity if both declarations are identical but for the fact that one version is able to accept a superset of the possible arguments that are acceptable for the other version. The original `sumVector()` template can accept only a pointer type as its first argument. The version declared here can accept a pointer type as well as any non-pointer type. A pointer type `iArray` is passed, so both template functions are candidates for instantiation. However, the original `sumVector()` template function can *only* accept a pointer type as its first argument. It is therefore more specialized than the one given here, and it is therefore selected by the compiler. If, for some reason, this is *not* appropriate, then an explicit template argument can be used to overrule the selection made by the compiler. E.g.,

```
sumVector<int *>(iArray, 12);
```

### 17.1.7 Selecting an overloaded (template) function

The following steps determine the actual function that is called, given a set of (template or non-template) overloaded functions:

- First, a set of candidate functions is constructed. This set contains all functions that are visible at the point of the call, having the same name as the function that is called. For a template function to be considered here, depends on the actual arguments that are used. These arguments must be acceptable given the standard template argument deduction process described in section 17.1.3. For example, assuming all of the following declarations were provided, an instantiation of

  ```
  template <class T, class U>
  bool differentSigns(T t, U u);
  ```

  and the functions

  ```
  bool differentSigns(double i, double j);
  bool differentSigns(bool i, bool j);
  bool differentSigns(int (&i)[2]);
  ```

  will all be elements of the set of possible functions in the following code fragment, as all of the four functions have the same name of the function that is called:

  ```
  void fun(int arg1, double arg2)
  {
      differentSigns(arg1, arg2);
  }
  ```

- Second, the set of *viable functions* is constructed. Viable functions are functions for which type conversions exist that can be applied to match the types of the parameters of the functions and the types of the actual arguments. This removes the last two function declarations from the initial set: the third function is removed as there is no standard conversion from `double` to `int`, and the fourth function is removed as there is a mismatch in the number of arguments between the called function and the declared function.

- Third, the remaining functions are ranked in order of preference, and the first one is going to be used. Let's see what this boils down to:

  For the template function, the function `differentSign<int, double>` is instantiated. For this function the types of the two parameters and arguments for a pairwise exact match: score two points for the template function.

  For the function `bool differentSigns(double i, double j)` the type of the second parameter is exactly matches the type of the second argument, but a (standard) conversion `int -> double` is required for the first argument: score one point for this function.

  Consequently, the template function is selected as the one to be used. As an exercise, feed the abouve four declarations and the function `fun()` to the compiler and wait for the linker errors: ignoring the undefined reference to `main()`, the linker will complain that the (template) function

  ```
  bool differentSigns<int, double>(int, double)
  ```

  is an undefined reference.

If the template would have been declared as

```
template <class T>
bool differentSigns(T t, T u);
```

then no template function would have been instantiated here. This is ok, as the ordinary function `differentSigns(double, double)` will now be used. An error occurs only if no instantiation of the template function can be generated and if no acceptable ordinary function is available. If such a case, the compiler will generate an error like

```
no matching function for call to ‘differentSigns (int &, double &)
```

As we've seen, a template function in which all type parameters exactly match the types of the arguments prevails over an ordinary function in which a (standard) type conversion is required. Correspondingly, a template explicitly specialized function will prevail over an instantiation of the general template if both instantiations show an exact match between the types of the parameters and the arguments. For example, if the following template declarations are available:

```
template <class T, class U>
bool differentSigns(T t, U u);

template <> bool differentSigns<double, int>(double, int);
```

then the template explicitly specialized function will be selected without generating an extra instantiation from the general template definition.

Another situation in which an apparent ambiguity arises is when both an ordinary function is available and a proper instantiation of a template can be generated, both exactly matching the types of the arguments of the called function. In this case the compiler does not flag an ambiguity as the oridinary function is considered the more specialized function, which is therefore selected.

As a rule of thumb consider that when there are multiple viable functions sharing the top ranks of the set of viable functions, then the function template instantiations are removed from the set. If only one function remains, it is selected. Otherwise, the call is ambiguous.

### 17.1.8 Name resolution within template functions

Consider once more our function `sumVector()` of section 17.1.1, but now it's given a somewhat different implementation:

```
template <class T>
T sumVector(T *array, unsigned n)
{
    T
        sum = accumulate(array, array + n, T(0));

    cout << "The array has " << n << " elements." << endl;
    cout << "The sum is " << sum << endl;

    return (sum);
}
```

In this template definition, `cout`'s `operator<<` is called to display a `char const *` string, an `unsigned`, and a `T`-value. The first `cout` statement displays the string and the `unsigned` value, no matter what happens in the template. These types *do not depend on a template parameter*. If a type does not depend on a template parameter, the necesary declarations for compiling the statement must be available when the definition of the template is given. In the above template definition this implies that

```
ostream &ostream::operator<<(unsigned)
```

and

```
ostream &ostream::operator<<(char const *)
```

must be known to the compiler when the definition of the template is given. On the other hand,

```
cout << ... << sum << endl
```

cannot be compiled by the time the template's definition is given, as the type of the variable `sum` *depends on a template parameter*. The statement can therefore be checked for semantical correctness (i.e., the question whether `sum` can be inserted into `cout`) can only be answered at the point where the template function is instantiated.

Names (variables) whose type depend on a template parameter are resolved when the template is instantiated: at that point the relevant declarations must be available. The location where this happens is called the template's *point of instantiation*. As a rule of thumb, make sure that the necessary declarations (usually: header files) are available at every instantiation of the template.

## 17.2 Template classes

Like templates for functions, templates can be constructed for complete classes. A template class can be considered when the class should be available for different types of data. Template classes are frequently used in **C++**: chapter 8 covers general data structures like `vector, stack` and `queue`, which are available as *template classes*. The algorithms and the data on which the algorithms operate are completely separated from each other. To use a particular data structure on a particular data type, only the data type needs to be specified at the definition or declaration of the template class object, e.g., `stack<int> istack`.

In the upcoming sections the construction of such a template class is discussed. In a sense, template classes compete with object oriented programming (cf. chapter 16), where a similar mechanism is seen. Polymorphism allows the programmer to separate algorithms from data, by deriving classes from the base class in which the algorithm is implemented, while implementing the data in the derived class, together with member functions that were defined as pure virtual functions in the base class to handle the data.

Generally, template classes are easier to use. It is certainly easier to write `stack<int> istack` to create a stack of `ints` than it is to derive a new class `Istack: public stack` and to implement all necessary member functions to be able to create a similar stack of `ints` using object oriented programming. On the other hand, for each different type that is used with a template class the complete class is reinstantiated, whereas in the context of object oriented programming the derived classes *use*, rather than *copy*, the functions that are already available in the base class.

Below a simple version of the template class `vector` is constructed: the essential characteristics of a template class are illustrated, without attempting to redo the existing `vector` class completely.

### 17.2.1 Template class definitions

The construction and use of template classes will be covered in the coming sections, where a basic template class `bvector` (*basic vector* will be constructed.

The construction of a template class can normally begin with the construction of a normal class interface around a hypothetical type `Type`. If more hypothetical types are required, then hypothetical types `U, V, W`, etc. can be used as well. Assume we want to construct a class `bvector`, that can be used to store values of type `Type`. We want to provide the class with the following members:

- Constructors to create an object of the class `bvector`, possibly of a given size, as well as a copy constructor, since memory will be allocated by the object to store the values of type `Type`.
- A destructor.
- An overloaded `operator=` operator.
- A `operator[]` to retrieve and reassign the elements giving their indices.

- Forward and backward iterators to be able to visit all elements sequentially, either from the first to the last or from the last to the first.
- A `sort()` member to sort the elements of type `Type`.
- A member `push_back()` to add a new element at the end of the vector.

Should the set of members include members that can be used with `const` objects? In practical situations it probably should, but for now these members are not included in the interface: I've left them for the reader to implement.

Now that we have decided which members we want, the class interface can be constructed. Like template functions, a template class definition begins with the keyword `template`, to be followed by a non-empty list of template *type* and/or *non-type* parameters, surrounded by angular brackets. This template announcement is then followed by the class interface, in which the template parameters may be used to represent types and constants. Here is initial class interface of the `bvector` template class, already showing member functions `construct` and `destroy` which are used in the implementation of the copy constructor, the destructor, and the overloaded assignment operator. The class also already contains an `iterator` type: it's defined simply as a pointer to an element of the vector. The reverse-iterator will be added later. Note that the `bvector` template class contains only a template type parameter, and no non-type parameter.

```
template <class Type>
class bvector
{
    public:
        typedef reverse_iter<Type> reverse_iterator;

        bvector();
        bvector(unsigned n);
        bvector(bvector<Type> const &other);
        ~bvector();
        bvector<Type> const &operator=(bvector<Type> const &other);
        Type &operator[](int index);
        bvector<Type> &sort();
        void push_back(Type const &value);
        Type *begin();
        Type *end();
        reverse_iterator rbegin();
        reverse_iterator rend();
        unsigned size();
    private:
        void construct(bvector<Type> const &other);
        Type
            *start,
            *finish,
            *end_of_storage;
};
```

Within the class interface definition the abstract type `Type` can be used as a normal typename. However, note the `bvector`<`Type`> constructions appearing in the interface: there is no plain `bvector`, as the `bvector` will be bound to a type `Type`, to be specified later on in the program using a `bvector`.

Different from template functions, template class parameters can have default arguments. This holds true both for template type- and template non-type parameters. If a template class is instantiated without specifying arguments for the template parameters, and if default template parameters were defined, then the defaults are used. Such defaults should be suitable for a majority of instantiations of the class. E.g., for the template class `bvector` the template announcement could have been altered to specify `int` as the default type:

```
template <class Type = int>
```

The class contains three data members: pointers to the begin and end of the allocated storage area (respectively `start` and `end_of_storage`) and a pointer pointing just beyond the element that was last allocated. The allocation scheme will add elements beyond the ones that are actually required to reduce the number of times the vector must be reallocated to accomodate new elements.

Template class *declarations* are constructed by removing the template interface definition (the part between the curly braces), replacing the definition by a semicolon:

```
    template <class Type>
    class bvector;
```

here too, default types may be specified.

In section 17.3 the full implementation of the `bvector` template class is given.

### 17.2.2 Template class instantiations

template classes are instantiated when an object of the template class is defined. When a template class object is defined (or declared) the template parameters must be explicitly specified (note that the parameters having default arguments are also specified, albeit as defaults). The template arguments are never deducted, as with template functions. To define a `bvector` to store `ints`, the construction

```
    bvector<int>
        bvInt;
```

is used. For a `bvector` for `strings`

```
    bvector<string>
        bvString;
```

is used.

In combination with the keyword `extern` these variables are *declared* rather than *defined*. E.g.,

```
extern bvector<int>
    bvInt;
```

A template (type) parameter can be used to designate a type within another template. In the following function the template function `manipulateVector()` is defined, using type parameter `T`. It receives, defines, and returns `bvector` references and objects:

```
template <class T>
bvector<T> &manipulateVector(bvector<T> &vector)
{
    bvector<T>
        extra(vector);
    ...
    return (vector);
}
```

A template class is not instantiated if a reference or pointer to the class template is used. In the above example, the `bvector<int> extra(...)` results in a template instantiation, but the parameter and the return value of the function `manipulateVector()`, being references, don't result in template instantiations. However, if a member function of a template class is used with a pointer or reference to a template class object, then the class *is* instantiated. E.g., in the following code

```
template <class T>
void add(bvector<T> &vector, int value)
{
    vector.push_back(value);
}
```

the class `bvector<int>` will be instantiated.

### 17.2.3 Nontype parameters

Template nontype parameters must be constant expressions. I.e., the compiler must be able to evaluate their values. For example, the following class uses a template type parameter to define the type of the elements of a buffer, and a template nontype parameter to define the size of the buffer:

```
template <class Type, unsigned size>
class Buffer
{
    ...
    Type
        buffer[size];
```

```
};
```

The `size` parameter must be a constant value when a `Buffer` object is defined or declared. E.g.,

```
Buffer<int, 20>
    buffer;
```

Note that

- Global variables have constant addresses, that can be used as arguments for nontype 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 nontype parameters.
- Lvalue transformations are allowed: if a pointer is defined as a nontype 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* datatype may be used for a nontype parameter of a *wider* type (e.g., `short` when an `int` is called for, `long` when a `double` is called for).
- Integral conversions are allowed: if an `unsigned` parameter is specified, an `int` may be used.

### 17.2.4 Template class member functions

Normal design considerations should be followed when constructing template class member functions or template class constructors: template class type parameters should preferably be defined as `T const &`, rather than `T`, to prevent unnecessary copying of large `T` types. Template class constructors should use member initializers rather than member assignment within the body of the constructors, again to prevent double assignment of composed objects: once by the default constructor of the object, once by the assignment itself.

Template member functions must be known to the compiler when the template is instantiated. The current *GNU g++* compiler does not allow precompiled template classes, therefore the member functions of templates are inline functions. They can be defined inside the template interface or outside the template interface. Template member functions are defined as the inline member functions of any other class. However, for the member functions that are defined outside of the template's interface

- No *inline* keyword is required in the interface,
- A `template <template parameter list>` definition is required.

In the bvector template class a member function

```
void push_back(T const &value);
```

is declared. Its definition, outside of the template's interface, could be:

```
template <class T>
void bvector<T>::push_back(T const &t)
{
    if (finish == end_of_storage)
    {
        end_of_storage <<= 1;
        T
            *tmp = copy(start, finish, new T[max]);
        delete [] start;
        finish = tmp + (finish - start);
        finish = tmp;
    }
    *finish++ = t;
}
```

Note the fact that the class type of `push_back` is the generic `bvector<T>` type. The abstract type `T` is also used to define the type of the variable `tmp`.

### 17.2.5 Template classes and friend declarations

Template classes may define other functions and classes as friends. There are three types of friend declarations that can appear within a template class:

- A *nontemplate* friend function or class. This is a well-known friend declaration.
- A *bound* friend template class or function. Here the template parameters of the current template are used to *bind* the types of another template class or function, so that a one-to-one correspondence between the template's parameters and the template parameters of the friend template class or function is obtained.
- A *unbound* friend template class or function. Here the template parameters of the friend template class or function remain to be specified, and are not related in some predefined way to the current template's parameters.

The following sections will discuss the three types of friend declarations in further detail.

#### Nontemplate friends

A template class may declare another function or class or class member function as its friend. Such a friend may access the private members of the template. Friend classes and ordinary friend functions can be declared as friends, but a class interface must have been seen by the compiler before one of its members can be declared a friend of a template class (in order to verify the name of the friend function against the interface.

For example, here are some friend declarations:

```
class Friend
{
    public:
        void member();
};

template <class T>
class bvector
{
    friend class AnotherFriend;     // declaration only is ok here
    friend void anotherMember();    // declaration is ok here
    friend Friend::member();        // Friend interface class required.
    ...
};
```

Such ordinary friends can be used, e.g., to access the static private members of the `bvector` class or they can themselves define `bvector` objects and access all members of these objects.

### Bound friends

With *bound friend* template classes or functions there is a one-to-one mapping between the types that are used with the instantiations of the friends and the template class declaring them as friends. Here the friends are themselves templates. For example:

```
template <class T>
class Friend;                   // declare a template class

template <class T>
void function(Friend<T> &t);    // declare a template function

template <class T>
class AnotherFriend
{
    public:
        void member();
}

template <class T>
class bvector
{
    friend class Friend<T>;                     // 1
    friend void function<T>(Friend<T> t);       // 2
    friend void AnotherFriend<T>::member();     // 3
};
```

Above, three friend declarations are defined:

- At 1, the class `Friend` is declared a friend of `bvector` if it is instantiated for the same type `T` as `bvector` itself.
- At 2, the function `funciont` is declared a friend of `bvector` if it is instantiated for the same type `T` as `bvector` itself. Note that the template type parameter `T` appears immediately following the function name in the friend declaration. Here the correspondence between the function's template parameter and `bvector`'s template parameter is defined. After all, `function()` could have been a parameterless function. Without the <T> affixed to the function name, it is an ordinary function, expecting an (unrestricted) instantiation of the class `bvector` for its argument.
- At 3, a specific member function of the class `AnotherFriend`, instantiated for type `T` is declared as a friend of the class `bvector`.

Assume we would like to be able to insert the elements of a `bvector` into an `ostream` object, using the insertion operator <<. For such a situation the `copy()` generic algorithm in combination with the `ostream_iterator` comes in handy. However, the latter iterator is a template function, depending on type `T`. If we can assume that `start` and `finish` are iterators of `bvector`, then the implementation is quickly realized by defining `operator<<` as a template function, and by declaring this operator as a friend of the class `bvector()`:

```
#include <iterator>
#include <algorithm>
#include <iostream>

template<class T>
class bvector
{
    friend ostream &operator<< <T> (ostream &str,
                                    bvector<T> const &vector);
    private:
        Iterator
            *start,
            *finish;
};

template <class T>
ostream &operator<<(ostream &str, bvector<T> const &vector)
{
    ostream_iterator<T *>
        out(str, " ");
    return (copy(bvector.start, bvector.finish, out));
}
```

### Unbound friends

By prepending the `friend` declarations by the `template<typelist>` phrase, the friends received their own template parameter list. The template types of these friends are completely independent from the type of the template class declaring the friends. Such friends are called *unbound friends*.

Every instantiation of an unbound friend has unrestricted access to the private members of every instantiation of the template class declaring the friends.

Here is the syntactic convention for declaring an unbound friend function, an unbound friend class and an unbound friend member function of a class:

```
template <class Type>
class bvector
{
    template <class T>
    friend void function(); // unbound friend function

    template <class T>
    friend class Friend;    // unbound friend class

    template <class T>      // unbound friend member function
    friend void AnotherFriend<T>::member();

    ...
};
```

Unbound friends may not yet be supported by your compiler, though. E.g., earlier versions of the `GNU g++` compiler )) used to complain with a message like

```
invalid member template declaration
```

However, current versions of the `egcs` compiler do accept unbound friends.

### 17.2.6 Template classes and static data

When static members are defined in a template class, these static members are instantiated for every different instantiation of the template class. As they are static members, there will be only one member when multiple objects of the *same* template type(s) are defined. For example, in a class like:

```
template <class Type>
class TheClass
{
    ...
    private:
        static int
            objectCounter;
};
```

There will be *one* `TheClass<Type>::objectCounter` for each different `Type`. However, the following will result in just one static variable, which is shared among the different objects:

```
TheClass<int>
    theClassOne,
    theClassTwo;
```

Remeber that static members are only *declared* in their classes. They must be *defined* separately. With static members of template classes this is not different. But, comparable to the implementations of static functions, the definitions of static members are usually provided in the same file as the template class interface itself. The definition of the static member objectCounter is therefore:

```
template <class Type>
class TheClass
{
    ...
    private:
        static int
            objectCounter;
};

template <class Type>
int
    TheClass<Type>::objectCounter = 0;
```

In the above case objectCounter is an int, and thus independent of the template type parameter Type. In a list-like construction, where a pointer to objects of the class itself is required, the template type parameter Type does enter the definition of the static variable, as is shown in the following example:

```
template <class Type>
class TheClass
{
    ...
    private:
        static TheClass
            *objectPtr;
};

template <class Type>
TheClass<Type>
    *TheClass<Type>::objectPtr = 0;
```

Note here that the definition can be read, as usual, from the variable name back to the beginning of the definition: objectPtr of the class TheClass<Type> is a pointer to an object of TheClass<Type>.

### 17.2.7 Derived Template Classes

Template classes can be used in class derivation as well. Consider the following base class:

```
template<class T>
class Base
{
    public:
        Base(T const &t)
        :
            t(t)
        {}
        // and other members
    private:
        T const &t;
};
```

The above class is a template class, which can be used as a base class for the following template class `Derived`:

```
template<class T>
class Derived: public Base<T>
{
    public:
        Derived(T const &t)
        :
            Base(t)
        {}
        // and other members
};
```

Other combinations are possible too: By specifying the template type parameters of the base class at the point where the base class is introduced as the base class of a derived class, the derived class becomes an ordinary (non-template) class:

```
class Ordinary: public Base<int>
{
    public:
        Ordinary(int x)
        :
            Base(x)
        {}
};

// With the following object definition:
Ordinary
```

```
    o(5);
```

### 17.2.8 Nesting and template classes

When a class is nested within a template class, it automatically becomes a template class itself. The nested class may use the template parameters of the surrounding class, as shown in the following small program:

```
#include <vector>

template<class Type>
class TheVector
{
    public:
        class Enumeration
        {
            public:
                Enumeration(vector<Type> const &vector)
                :
                    vp(&vector),
                    idx(0)
                {
                }
                Type const &nextElement()   // uses 'Type'
                {
                    if (idx == vp->size())
                        throw NoSuchElementException(index);
                    return ((*vp)[idx++]);
                }
                bool hasMoreElements()
                {
                    return (idx < vp->size());
                }
            private:
                vector<Type>
                    const *vp;
                unsigned
                    idx;
        };

        TheVector<Type>::Enumeration getEnumeration()
        {
            return (Enumeration(vector));
        }
    private:
        vector<Type>
            vector;
};

int main()
```

```
{
    TheVector<int>
        theVector;

    TheVector<int>::Enumeration
        en = theVector.getEnumeration();

    cout << (en.hasMoreElements() ? "has more elements" :
                                    "no more elements") << endl;

    return (0);
}
```

In the above program the class Enumeration is a nested class, that uses the template parameter Type of its surrounding class. The nested class Enumeration defines an object that returns the subsequent elements of the vector of the surrounding class, and allows a simple query about the existence of another element.

(Parts of) the nested class are instantiated once used. E.g., in the above example, the function nextElent() is not used. This is why the example can be compiled to a working program, as the NoSuchElementException() exception was never defined!

Enumerations and typedefs can be defined nested in template classes as well. For example, with arrays the distinction between the last index that can be used and the number of elements frequently causes confusion in people who are first exposed to the **C**-array types. The following construction automatically provides a valid last and nElements definition:

```
template<class Type, int size>
class Buffer
{
    public:
        enum Limits
        {
            last = size - 1,
            nElements
        };
        typedef Type elementType;

        Buffer()
        :
            b(new Type [size])
        {}
    private:
        Type
            *b;
};
```

This small example defines Buffer<Type, size>::elementType, Buffer<Type, size>::last and Buffer<Type, size>::nElements (as values), as well as Buffer<Type, size>::Limits and Buffer<Type, size>::elementType> (as typenames).

Of course, the above represents the template form of these values and declarations. They must be instantiated before they can be used. E.g,

`Buffer<int, 80>::elementType`

is a synonym of `int`.

Note that a construction like `Buffer::elementType` is illegal, as the type of the `Buffer` class remains unknown.

### 17.2.9 Template members

It is possible to define a template class or a template function within another class (which itself may or may not be a template class). Such a template function or template class is called a *member template*. It is defined as any other ordinary template class, including the `template <class ...>` header. E.g.,

```
template <class T>
class Outer
{
    public:
        ...
        template <class T2>     // template class
        class Inner
        {
            public:
                T
                    tVariable;
                T2
                    t2Variable;
        };
        template <class Type>
        Type process(Type const &p1, Type const &p2)
        {
            Type
                result;
            ...
            return (result);
        }
        ...
};
```

The special characteristic of a member template is that it can use its own and its surrounding class' template parameters, as illustrated by the definition of `tVariable` in `Inner`.

Normall access rules apply: the function `process()` can be used by the general program, given an instantiated `Outer` object. Of course, this implies that a large number of possible instantiations of `process()` are possible. Actually, an instantiation is only then constructed when a `process()` function is in fact used. In the following code the function member function `int process(int const &p1, int const &p2)` is instantiated, even though the object is of the class `Outer<double>`:

```
Outer<double>
    outer;

outer.process(10, -3);
```

The template member function allows the processing of any other type by an object of the class `Outer`, which becomes important if the other type can be converted to the type that's used by the outer template class.

Any function can be defined as a template function, not just an ordinary member function. A constructor can be defined as a template as well:

```
template <class T>
class Outer
{
    public:
        template <class T2>     // template class
        Outer(T2 const &initialValue)
        {
            ...
        }
        ...
};
```

Here, an `Outer` object can be constructed for a particular type given another type that's passed over to the constructor. E.g.

```
Outer<int>
    t(12.5);    // uses Outer(double const &initialvalue)
```

Template members can be defined inline or outside of their containing class. When a member is defined outside of its surrounding class, the template parameter list must precede the template parameter list of the template member. E.g.,

```
template <class T>
class Outer
{
    public:
        template <class T2>     // template class
        class Inner;

        template <class Type>
        Type process(Type const &p1, Type const &p2);
};
```

```
template <class T> template <class Type>    // template class member
class Outer<T>::Inner<Type>
{
    public:
        T
            tVariable;
        T2
            t2Variable;
};

template <class T> template <class Type>    // template function member
Type Outer<T>::process(Type const &p1, Type const &p2)
{
    Type
        result;
    ...
    return (result);
}
```

Not all compilers fully support member templates yet. E.g., the current `GNU g++` compiler (2.95.2) does not support the member template classes, but it does support the member template functions.

### 17.2.10 Template class specializations

Template class specializations are used in cases where template member functions cannot be used with a (class) type for which the template is instantiated. In those cases the template's member function(s) can be explicitly constructed to suit the needs of the particular type for which the template is instantiated.

Assume we have a template class which supports the insertion of its type parameter into an ostream. E.g.,

```
template <class Type>
class Inserter
{
    public:
        Inserter(Type const &t)
        :
            object(t)
        {}
        ostream &insert(ostream &os) const
        {
            return (os << object);
        }
    private:
        Type
            object;
};
```

In the example a plain member function is used to insert the current object into an ostream. The implementation of the **insert()** function shows that it uses the **operator**<<, as defined for the type that was used when the template class was instantiated. E.g., the following little program instantiates the class **Inserter<int>**:

```
int main()
{
    Inserter<int>
        ins(5);

    ins.insert(cout) << endl;
    return (0);
}
```

Now suppose we have a **class Person** having, among other members, the following member function:

```
class Person
{
    public:
        ostream &insert(ostream &ostr) const;
};
```

This class cannot be used to instantiate **Inserter**, as it does not have a **operator<<()** function, which is used by the function **Inserter<Type>::insert()**. Attempts to instantiate **Inserter<Person>** will result in a compilation error. For example, consider the following **main()** function:

```
int main()
{
    Person
        person;
    Inserter<Person>
        p2(person);

    p2.insert(cout) << endl;
}
```

If this function is compiled, the compiler will complain about the missing function **ostream & << const Person &**, which was indeed not available. However, the function **ostream &Person::insert(ostream &ostr)** *is* available, and it serves the same purpose as the required function **ostream & Inserter<Person>::insert(ostr &)**.

For this situation multiple solutions exist. One would be to define an **operator<<(Person const &p)** function which calls the **Person::insert()** function. But in the context of the **Inserter** class, this might not what we want. Instead, we might want to look for a solution that is closer to the `class Inserter`.

Such a solution exists in the form of a *template class specialization*. Such an *explicit specialization definition* starts with the wordt `template`, then two angular brackets (<>), which is then followed by the function definition for the instantiation of the template class for the particular template parameter(s). So, with the above function this yields the following function definition:

```
template<>
ostream  &Inserter<Person>::insert(ostream &os) const
{
    return (object.insert(os));
}
```

Here we explicitly define a function `insert` of the class `Inserter<Person>`, which calls the appropriate function that lives in the `Person` class.

Note that the explicit specialization definition is a true *definition*: it should *not* be given in the header file of the `Inserter` template class, but it should have its own sourcefile. However, in order to inform the compiler that an explicit specialization is available, it can be *declared* in the template's header file. The declaration is straightforward: the code-block is replaced by the semicolon:

```
template<>
ostream  &Inserter<Person>::insert(ostream &os) const;
```

It is even possible to specialize a complete template class. For the above class `Inserter` which would boil down to the following for the class `double`:

```
template <>
class Inserter
{
    public:
        Inserter<double>(double const &t);
        ostream &insert(ostream &os) const;
    private:
        double
            object;
};
```

The explicit template class specialization is obtained by replacing all references to the template's class name `Inserter` by the class name and the type for which the specialization holds true: `Inserter<double>`, and by replacing occurrences of the template's type parameter by the actual type for which the specialization was constructed. The complete template class specialization interface must be given *after* the original template class has been defined. The definition of its members are, analogously to the `Inserter<Person>::insert()` function above, given in separate source files. However, in the case of a complete template class specialization, the definitions of its members should *not* be preceded by the **template<>** prefix. E.g.,

```
Inserter<double>(double const &t)   // NO template<> prefix !
:
    object(t)
{}
```

### 17.2.11 Template class partial specializations

In cases where a template has more than one parameter, a *partial specialization* rather than a full specialization might be appropriate. With a `partial` specialization, a subset of the parameters of the original template can be redefined.

Let's assume we are working on a image processing program. A class defining an image receives two `int` template parameters, e.g.,

```
template <int columns, int rows>
class Image
{
    public:
        Image()
        {
            // use 'columns' and 'rows'
        }
        ...
};
```

Now, assume that an image having 320 columns deserves special attention, as those pictures require, e.g., a special smoothing algorithm. From the general template given above we can now construct a *partially specialized* template, which *only* has a `columns` parameter. Such a template is like an ordinary template parameter, in which only the `rows` remain as a template parameter. At the definition of the class name the specialization is made explicit by mentioning a specialization parameter list:

```
template <int rows>
class Image<320, rows>
{
    public:
        Image()
        {
            // use 320 columns and 'rows' rows.
        }
        ...
};
```

With the above partially specialized template definition the 320 columns are explicitly mentioned at the class interface, while the `rows` remain variable. Now, if an image is defined as

```
Image<320, 240>
    image;
```

two instantiations *could* be used: the fully general template is a candidate as well as the partially specialized template. Since the partially specialized template is *more specialized* than the fully general template, the `Image<320, rows>` template will be used. This is a general rule: a more specialized template instantiation is chosen in favor of a more general one whereever possible.

Every template parameter can be used for the specialization. In the last example the `columns` were specialized, but the `rows` could have been specialized as well. The following partial specialization of the template class `Image` specializes the `rows` parameter and leaves the `columns` open for later specification:

```
template <int columns>
class Image<columns, 200>
{
    public:
        Image()
        {
            // use 'columns' columns and 200 rows.
        }
        ...
};
```

Even when *both* specializations are provided there will (generally) be no problem. The following three images will result in, respectively, an instantiation of the general template, of the template that has been specialized for 320 columns, and of the template that has been specialized for the 200 rows:

```
Image<1024, 768>
    generic;
Image<320, 240>
    columnSpecialization;
Image<480, 200>
    rowSpecialization;
```

With the `generic` image, no specialized template is available, so the general template is used. With the `columnSpecialization` image, 320 columns were specified. For that number of columns a specialized template is available, so it's used. With the `rowSpecialization` image, 200 rows were specified. For that number of rows a specialized template is available, so that specialized template is used with `rowSpecialization`.

One might wonder what happens if we want to construct a

```
Image<320, 200>
    superSpecialized;
```

image. Is this a specialization of the columns or of the rows? The answer is: neither. It's an ambiguity, precisely because *both* the columns *and* the rows could be used with a (differently) specialized template. If such an image is required, yet another specialized template is needed, albeit that that one isn't a *partially* specialized template anymore. Instead, it specializes all its parameters with the class interface:

```
template <>
class Image<320, 200>
{
    public:
        Image()
        {
            // use 320 columns and 200 rows.
        }
        ...
};
```

The above super specialization of the `Image` template will be used with the image having 320 columns and 200 rows.

### 17.2.12 Name resolution within template classes

In section 17.1.8 the name resolution process with template functions was discussed. As is the case with template functions, name resolution in template classes also proceeds in two steps. Names that do not depend on template parameters are resolved when the template is defined. E.g., if a member function in a template class uses a `qsort()` function, then `qsort()` does not depend on a template parameter. Consequently, `qsort()` must be known when the compiler sees the template definition, e.g., by including the file `stdlib.h`.

On the other hand, if a template defines a <`class Type`> template parameter, which is the returntype of some template function, e.g.,

```
Type returnValue();
```

then we have a different situation. At the point where template objects are defined or declared, at the point where template member functions are used, and at the point where static data members of template classes are defined or declared, it must be able to resolve the template type parameters. So, if the following template class is defined:

```
template <class Type>
```

```
class Resolver
{
    public:
        Resolver();
        Type result();
    private:
        Type
            datum;
        static int
            value;
};
```

Then `string` must be known before each of the following examples:

```
// ----------------------- example 1: define the static variable
int Resolver<string>::value = 12;

// ----------------------- example 2: define a Resolver object
int main()
{
    Resolver<string>
        resolver;
}

// ----------------------- example 3: declare a Resolver object
extern Resolver<string>
    resolver;
```

## 17.3 An example: the implementation of the bvector template

In this section the implementation of the basic vector `bvector`, introduced in section 17.2.1, will be completed.

The implementation of the `bvector` is generally straightforward: the basic constructors initialize the data members of the `bvector`, using an auxiliary private function `init()`:

```
bvector()
{
    init(0);
};
bvector(unsigned n)
{
    init(n);
}
void init(unsigned n)
```

```
{
    if (n)
    {
        start = new Type[n];
        finish = start + n;
        end_of_storage = start + n;
    }
    else
    {
        start = 0;
        finish = 0;
        end_of_storage = 0;
    }
}
```

The copy-constructor, overloaded assignment operator and destructor are also constructed according to a general recipe. The destructor is simple: it only has to call the operator `delete` for `start`, using the `[]` notation to make sure that class objects stored in the `bvector` are deleted too. Therefore, no `destroy()` function was considered necessary in this class. Note that storing pointers in the `bvector` is dangerous, as it is with the official `STL vector` type: the data pointed to by pointer elements of `bvector` is not deleted when the `bvector` itself is destroyed.

Here are the destructor, the copy constructor, the overloaded assignment operator and the private `construct()` function:

```
~bvector()
{
    delete [] start;
}

bvector(bvector<Type> const &other)
{
    construct(other);
}

bvector<Type> const &operator=(bvector<Type> const &other)
{
    if (this != &other)
    {
        delete [] start;
        construct(other);
    }
    return (*this);
}

void construct(bvector<Type> const &other)
{
    init(other.finish - other.start);
    copy(other.start, other.finish, start);
}
```

The `operator[]` first checks the validity of the `index` that's passed to the function. If out of bounds a simple exception is thrown. Otherwise the function is completely standard. Note that the current implementation of `bvector` does not allow for `bvector<Type> const` objects to use the `operator[]`. Here is the implementation of the `operator[]` function:

```
Type &operator[](unsigned index) throw(char const *)
{
    if (index > (finish - start))
        throw "bvector array index out of bounds";
    return (start[index]);
}
```

The `sort()` function uses the available `sort()` generic algorithm. The `::sort()` notation is required to prevent confusion: without the scope resolution operator the compiler complains about us having specified the wrong arguments for the function `sort()`. Here is the implementation of `sort()`:

```
bvector<Type> &sort()
{
    ::sort(start, finish);
    return (*this);
}
```

The *push_back()* function either initializes the size of the `bvector` to one element, or doubles the number of elements in the vector when there's no more room to store new elements. When the number of elements must be doubled, an auxiliary `bvector` object is created, into which the elements of the current `bvector` object are copied, using the `copy()` generic algorithm. Next, the memory pointed to by the the current `bvector` object is deleted, and its pointers are reassigned to point to the memory occupied by the auxiliary `bvector` object. The `start` pointer of the auxiliary `bvector` object is then set to 0, to prevent the destruction of its memory, to which the current `bvector` points as well. Finally the new value is stored in the vector. Here is the implementation:

```
void push_back(Type const &value)
{
    if (!finish)
    {
        init(1);
        finish = start;
    }
    else if (finish == end_of_storage)
    {
        bvector<Type>
            enlarged((end_of_storage - start) << 1);
        copy(start, finish, enlarged.start);
        delete [] start;
        finish = enlarged.start + (finish - start);
```

```
            start = enlarged.start;
            end_of_storage = enlarged.end_of_storage;
            enlarged.start = 0;
        }
        *finish++ = value;
    }
```

Two sets of iterators are available: the `begin()` and `end()` functions return `iterators`, the `rbegin()` and `rend()` functions return `reverse_iterators`. `Iterators` and `reverse_iterators` are defined as `typedefs` within the template class. These typedefs and the functions returning the (reverse) iterators are given below:

```
    typedef Type *iterator;
    typedef reverse_iter<Type> reverse_iterator;

    iterator begin()
    {
        return (start);
    }
    iterator end()
    {
        return (finish);
    }
    reverse_iterator rbegin()
    {
        return (reverse_iterator(finish));
    }
    reverse_iterator rend()
    {
        return (reverse_iterator(start));
    }
```

The `iterator` is simply a type definition for a pointer to a `Type`. The `reverse_iterator` is more complex, as its type definition depends on a `reverse_iter<iterator>` type, defining the actual reverse iterator. The `reverse_iter<iterator>` itself is a template class, that is discussed in the next section.

### 17.3.1 The reverse_iter template class

The template class `reverse_iter` uses a template class parameter `Type` representing the data type for which a reverse iterator must be constructed. Since the type of the data to which the reverse iterator points is known, a reference and a pointer to the data type can easily be constructed.

Given the data type `Type` to which a reverse iterator points, the reverse iterator must support the following operations:

- It must be possible to construct a reverse iterator from an iterator.

- A dereference operator `Type &operator*()`, returning the data item to which the reverse iterator points.
- A pointer operator `Type *operator->()` returning the address of the data element, to be used when the data element is an object having members.
- A prefix and postfix increment operator returning a reverse iterator pointing to the *previous* data element.

As the reverse iterator returns a pointer to the previous element, it is possible to let the `rbegin()` iterator return a pointer to the last element, and to let `rend()` return a pointer to the address before the first data element. But it is also possible to let `rbegin()` return `end()`, and to let `rend()` return `begin()`. That way the pointers are used the same way, both for iterators and reverse iterators. This latter approach, which is used by the standard template library's implementation of the reverse iterators, requires the dereference operator to return the data element *before* the one to which the reverse iterator actually points. The implementation of the `operator*()` is, therefore:

```
Type &operator*() const
{
    Type
        *tmp = current;
    return (*--tmp);
}
```

The increment operators return reverse iterators. The prefix increment operator reduces the `current` pointer, and returns a reference to the current reverse iterator by returning `*this`:

```
reverse_iter<Type>& operator++()
{
    --current;
    return (*this);
}
```

The postfix increment operator returns a reverse iterator object which is a copy of the current reverse iterator, whose pointer `current` is reduced by applying the postfix decrement operator on the `current` pointer:

```
reverse_iter<Type> operator++(int)
{
    reverse_iter<Type>
        tmp(current--);
    return (tmp);
}
```

Of course, the `operator+(int step)` and the `operator--()` could be defined as well. These definitions are left as an exercise for the reader.

### 17.3.2 The final implementation

Below is the implementation of the template class `bvector` and its auxiliary template class `reverse_iter`:

```
#include <algorithm>

template <class Type>
class reverse_iter
{
    public:
        explicit reverse_iter(Type *x)
        :
            current(x)
        {}
        Type &operator*() const
        {
            Type
                *tmp = current;
            return (*--tmp);
        }
        Type *operator->() const
        {
            return &(operator*());
        }
        reverse_iter<Type>& operator++()
        {
            --current;
            return (*this);
        }
        reverse_iter<Type> operator++(int)
        {
            reverse_iter<Type>
                tmp(current--);
            return (tmp);
        }
        bool operator!=(reverse_iter<Type> const &other)
        {
            return (current != other.current);
        }
    private:
        Type
            *current;
};

template <class Type>
class bvector
{
    typedef Type *iterator;
    typedef reverse_iter<Type> reverse_iterator;

    public:
        bvector()
```

```
{
    init(0);
};
bvector(unsigned n)
{
    init(n);
}
bvector(bvector<Type> const &other)
{
    construct(other);
}
~bvector()
{
    delete [] start;
}
bvector<Type> const &operator=(bvector<Type> const &other)
{
    if (this != &other)
    {
        delete [] start;
        construct(other);
    }
    return (*this);
}
Type &operator[](unsigned index) throw(char const *)
{
    if (index >= (finish - start))
        throw "bvector array index out of bounds";
    return (start[index]);
}
bvector<Type> &sort()
{
    ::sort(start, finish);
    return (*this);
}
void push_back(Type const &value)
{
    if (!finish)
    {
        init(1);
        finish = start;
    }
    else if (finish == end_of_storage)
    {
        bvector<Type>
            enlarged((end_of_storage - start) << 1);
        copy(start, finish, enlarged.start);
        delete [] start;
        finish = enlarged.start + (finish - start);
        start = enlarged.start;
        end_of_storage = enlarged.end_of_storage;
        enlarged.start = 0;
    }
```

```
            *finish++ = value;
        }
        iterator begin()
        {
            return (start);
        }
        iterator end()
        {
            return (finish);
        }
        reverse_iterator rbegin()
        {
            return (reverse_iterator(finish));
        }
        reverse_iterator rend()
        {
            return (reverse_iterator(start));
        }
        unsigned size()
        {
            return (finish - start);
        }
    private:
        void init(unsigned n)
        {
            if (n)
            {
                start = new Type[n];
                finish = start + n;
                end_of_storage = start + n;
            }
            else
            {
                start = 0;
                finish = 0;
                end_of_storage = 0;
            }
        }
        void construct(bvector<Type> const &other)
        {
            init(other.finish - other.start);
            copy(other.start, other.finish, start);
        }

        Type
            *start,
            *finish,
            *end_of_storage;
};
```

A small `main()` function using the `bvector` data type is given next:

```
#include <iostream>
#include <string>
#include "bvector.h"

int main()
{
    bvector<int>
        bv(5),
        b2;

    b2 = bv;

    bv[0] = 3;
    bv[1] = 33;
    bv[2] = 13;
    bv[3] = 6;
    bv[4] = 373;

    copy(bv.begin(), bv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    bvector<int>::reverse_iterator
            rit  = bv.rbegin();

    while (rit != bv.rend())
        cout << *rit++ << ", ";
    cout << endl;

    bv.push_back(12);
    bv.push_back(5);

    copy(bv.begin(), bv.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    bv.sort();
    copy(bv.begin(), bv.end(), ostream_iterator<int>(cout, " "));
    cout << "bv has " << bv.size() << " elements\n";

    bvector<string>
        bstr;

    bstr.push_back("bravo");
    bstr.push_back("delta");
    bstr.push_back("foxtrot");
    bstr.push_back("echo");
    bstr.push_back("charley");
    bstr.push_back("alpha");

    bstr.sort();
    copy(bstr.begin(), bstr.end(), ostream_iterator<string>(cout, " "));
```

```
    cout << endl;
}
```

# Chapter 18

# Concrete examples of C++

This chapter presents a number of concrete examples of programming in **C++**. Items from this document such as virtual functions, `static` members, etc. are rediscussed. Examples of container classes are shown.

Another example digs into the peculiarities of using a parser- and scanner-generator with **C++**. Once the input for a program exceeds a certain level of complexity, it's advantageous to use a scanner- and parser-generator for creating the code which does the actual input recognition. The example describes the usage of these tool in a **C++** environment.

## 18.1 Storing objects: Storable and Storage

A reoccurring task of many programs is the storage of data, which are then sorted, selected, etc.. Storing data can be as simple as maintaining an array of `int`s, but can also be much more complex, such as maintaining file system information by the kernel of an operating system.

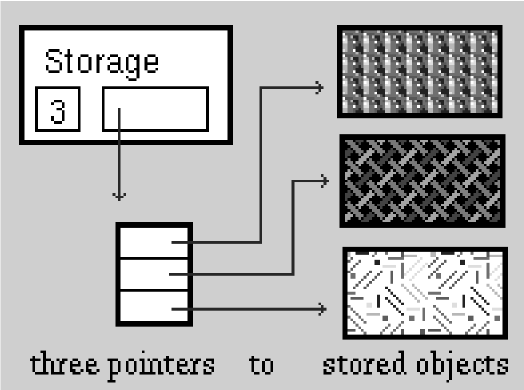
In this section we take a closer look at the storage of generic objects in memory (i.e., during the execution of a program). Conforming to the object-oriented recipe we shall develop two classes: a class `Storage`, which stores objects, and a class `Storable`, the prototype of objects which can be stored.

### 18.1.1 The global setup

As far as the functionality of the class `Storage` is concerned, objects can be added to the storage and objects can be obtained from the storage. Also it must be possible to obtain the number of objects in the storage.

As far as the internal data organization of the storage is concerned, we opt for an approach in which `Storage` maintains an array which can be reallocated, consisting of pointers to the stored objects.

The internal organization of the class `Storage` is illustrated in figure 18.1.




\

Figure 18.1: Internal organization of the class `Storage`.

### Interface functions of the class Storage

The usage (interface) of the class `Storage` is contained in three member functions. The following list describes these member functions and mentions the class `Storable`, more on this later.

- The function `add(Storable const *newobj)` adds an object to the storage. The function reallocates the array of pointers to accommodate one more and inserts the address of the object to store.
- The function `Storable const *get(int index)` returns a pointer to the object which is stored at the index'th slot.
- The function `int nstored()` returns the number of objects in the storage.

### To copy or not to copy?

There are two distinct design alternatives for the function `add()`. These considerations address the choice whether the stored objects (the squares on the right side of figure 18.1) should be **copies** of the original objects, or the objects **themselves**.

In other words, should the function `add()` of the class `Storage`:

- just store the address of the object which it receives as its argument in the array of pointers, or should it
- make a copy of the object first, and store the address of the copy?

These considerations are not trivial. Consider the following example:

```
Storage
    store;
Storable
    something;
```

```
store.add(something);           // add to storage

// let's assume that Storable::modify() is defined
something.modify();     // modify original object,

Storable
    *retrieved = store.get(0); // retrieve from storage

// NOW: is "*retrieved" equal to "something" ?!
```

If we choose to store (addresses of) the objects themselves, then at the end of the above code fragment, the object pointed to by `retrieved` will equal `something`. A manipulation of previously stored objects thereby alters the contents of the storage.

If we choose to store **copies** of objects, then obviously *`retrieved` will not equal `something` but will remain the original, unaltered, object. This approach has a great merit: objects can be placed into storage as a 'safeguard', to be retrieved later when an original object was altered or even ceased to exist. In this implementation we therefore choose for this approach.

### Who makes the copy?

The fact that copies of objects should be stored presents a small problem. If we want to keep the class `Storage` as universal as possible, then the making of a copy of a `Storable` object cannot occur here. The reason for this is that the actual type of the objects to store is not known in advance. A simplistic approach, such as the following:

```
void Storage::add(Storable const *obj)
{
    Storable
        *to_store = new Storable(*obj);
    // now add to_store instead of obj
    .
    .
}
```

shall not work. This code attempts to make a copy of `obj` by using the operator `new`, which in turn calls the copy constructor of `Storable`. However, if `Storable` is only a base class, and the class of the object to store is a derived class (say, a `Person`), how can the copy constructor of the class `Storable` create a copy of a `Person`?

The making of a copy therefore **must** lie with the actual class of the object to store, i.e., with the derived class. Such a class must have the functionality to create a **duplicate** of the object in question and to return a pointer to this duplicate. If we call this function `duplicate()` then the code of the adding function becomes:

```
void Storage::add(Storable const *obj)
{
    Storable
        *to_store = obj->duplicate();
    // now add to_store instead of obj
    .
    .
}
```

The function `duplicate()` is called in this example by using a pointer to the original object (this is the pointer `obj`). The class `Storable` is in this example only a base class which defines a protocol, and not the class of the actual objects which will be stored. Ergo, the function `duplicate()` need not be defined in `Storable`, but **must** be concretely implemented in derived classes. In other words, `duplicate()` is a **pure virtual** function.

### 18.1.2 The class Storable

Using the above discussed approach we can now define the class `Storable`. The following questions are of importance:

- Does the class `Storable` need a default constructor, or possibly other constructors such as a copy constructor?

  The answer is no. `Storable` will be a bare prototype, from which other classes will be derived.

- Does the class `Storable` need a destructor? Should this destructor be (pure) `virtual`?

  Yes. The destructor will be called when, e.g., a `Storage` object ceases to exist. It is quite possible that classes which will be derived from `Storable` will have their own destructors: we should therefore define a `virtual` destructor, to ensure that when an object pointed to by a `Storable*` is deleted, the actual destructor of the derived class is called.

  The destructor however should not be **pure** virtual. It is quite possible that the classes which will be derived from `Storable` will not need a destructor; in that case, an empty destructor function should be supplied.

The class definition and its functions are given below:

```
class Storable
{
    public:
        virtual ~Storable();
        virtual Storable *duplicate() const = 0;
};

Storable::~Storable()
{
}
```

### Converting an existing class to a Storable

To show how (existing) classes can be converted to derivation from a `Storable`, consider the below class `Person` from section 6.1. This class is re-created here, conforming to `Storable`'s protocol (only the relevant or new code is shown):

```
class Person: public Storable
{
    public:
        // copy constructor
        Person(Person const &other);

        // assignment
        Person const &operator=(Person const &other);

        // duplicator function
        Storable *duplicate() const;

    .
    .
};
```

When implementing the function `Person::duplicate()` we can use either the copy constructor or the default constructor with the overloaded assignment operator. The implementation of `duplicate()` is quite simple:

```
// first version:
Storable *Person::duplicate() const
{
    // uses default constructor in new Person
    Person
        *dup = new Person;

    // uses overloaded assignment in *dup = *this
    *dup = *this;

    return (dup);
}

// second version:
Storable *Person::duplicate() const
{
    // uses copy constructor in new Person(*this)
    return (new Person(*this));
}
```

The above conversion from a class `Person` to the needs of a `Storable` supposes that the sources of `Person` are at hand and can be modified. However, even if the definition of a `Person` class is not available, but is e.g., contained in a run-time library, the conversion to the `Storable` format poses no difficulties:

```
class StorablePerson: public Person, public Storable
{
    public:
        // duplicator function
        Storable *duplicate() const;
};

Storable *StorablePerson::duplicate() const
{
    return (new StorablePerson(*this));
}
```

### 18.1.3 The class Storage

We can now implement the class `Storage`. The class definition is given below:

```
class Storage: public Storable
{
    public:
        // destructors, constructor
        ~Storage();
        Storage();
        Storage(Storage const &other);

        // overloaded assignment
        Storage const &operator=(Storage const &other);

        // functionality to duplicate storages
        Storable *duplicate() const;

        // interface
        void add(Storable *newobj);
        int nstored() const;
        Storable *get(int index);

    private:
        // copy/destroy primitives
        void destroy();
        void copy(Storage const &other);

        // private data
        int n;
        Storable **storage;
};
```

Concerning the class definition we remark:

- As its interface the class has the functions `add()`, `get()` and `nstored()`. These functions were previously discussed (see section 18.1.1).
- The class has a copy constructor and an overloaded assignment function. These functions are needed because `Storage` contains a pointer, which addresses allocated memory.
- `Storage` itself is derived from `Storable`, as can be seen in the classname definition and in the presence of the function `duplicate()`. This means that `Storage` objects can themselves be placed in a `Storage`, thereby creating 'super-storages': say, a list of groups of `Person`s.
- Internally, `Storage` defines two `private` functions `copy()` and `destroy()`. The purpose of these primitive functions is discussed in section 6.4.1.

The destructor, constructors and the overloaded assignment function are listed below:

```
// default constructor
Storage::Storage()
{
    n = 0;
    storage = 0;
}

// copy constructor
Storage::Storage(Storage const &other)
{
    copy(other);
}

// destructor
Storage::~Storage()
{
    destroy();
}

// overloaded assignment
Storage const &Storage::operator=(Storage const &other)
{
    if (this != &other)
    {
        destroy();
        copy(other);
    }
    return (*this);
}
```

The primitive functions `copy()` and `destroy()` unconditionally copy another `Storage` object, or destroy the contents of the current one. Note that `copy()` calls `duplicate()` to duplicate the other's stored objects:

```
void Storage::copy(Storage const &other)
{
    n = other.n;
    storage = new Storable* [n];
    for (int i = 0; i < n; i++)
        storage [i] = other.storage [i]->duplicate();
}

void Storage::destroy()
{
    for (register int i = 0; i < n; i++)
        delete storage [i];
    delete storage;
}
```

The function `duplicate()`, which is required since `Storage` itself should be a `Storable`, uses the copy constructor to duplicate the current object:

```
Storable *Storage::duplicate() const
{
    return (new Storage (*this));
}
```

Finally, here are the interface functions which add objects to the storage, return them, or determine the number of stored objects[1]

```
void Storage::add(Storable const *newobj)
{
    // reallocate storage array
    storage = (Storable **) realloc(storage,
                (n + 1) * sizeof(Storable *));
    // put duplicate of newobj in storage
    storage [n] = newobj->duplicate();
    // increase number of obj in storage
    n++;
}

Storable *Storage::get(int index)
{
    // check if index within range
    if (index < 0 || index >= n)
        return (0);
    // return address of stored object
    return (storage [index]);
```

[1]Note: the function `realloc()` that is used in this section should actually not be used. A better procedure would be to create a **C++** variant for the `realloc()` function. A modification is in the pipeline....

```
}

int Storage::nstored() const
{
    return (n);
}
```

## 18.2 A binary tree

This section shows an implementation of a binary tree in **C++**. Analogously to the classes `Storage` and `Storable` (see section 18.1) two separate classes are used: one to represent the tree itself, and one to represent the objects which are stored in the tree. The classes will be appropriately named `Tree` and `Node`.

### 18.2.1 The Node class

The class `Node` is an abstract (pure virtual) class, which defines the protocol for the usage of derived classes with a `Tree`. Concerning this protocol we remark the following:

- When data are stored in a binary tree, the place of the data is determined by some **order**: it is necessary to determine how the objects should be sorted. This requires a comparison between objects. This comparison must inform the caller (i.e., the function which places objects in a tree) whether one object is 'smaller' or 'greater' than another object.

  This comparison must lie with `Node`s: a `Tree` itself cannot know how objects should be compared. Part of the procotol which is required by `Node` is therefore:

  ```
  virtual int compare(Node const &other) const = 0;
  ```

  The comparing function will have to be implemented in each derived class.

- Similar to the storage of objects in the class `Storage` (see section 18.1), a binary tree will contain **copies** of objects. The responsibility to duplicate an object therefore also lies with `Node`, as enforced by a pure virtual function:

  ```
  virtual Node *clone() const = 0;
  ```

- When processing a binary tree containing objects, the tree is recursively descended and a given operation is performed for each object. The operation depends of course on the actual type of the stored object. By declaring a pure virtual function

  ```
  virtual void process() = 0;
  ```

  in the class `Node`, the responsibility to process an object is placed with the derived class.

- When an object is to be stored in a binary tree, it may be that the object had already been stored previously. In that case the object will not be stored twice.

  For these cases we define a virtual function `rejected()`, which is a virtual function called for the `Node` that was already stored, receiving the node requesting to be added as its argument. However, since it's a virtual function, it can be redefined in a derived class:

```
    virtual void rejected(Node const &twice)
    {
    }
```

The complete definition and declaration of the class `Node` is given below:

```
class Node
{
    public:
        virtual ~Node()                     // destructor
        {}
        virtual Node* clone() const = 0;    // duplicator
        virtual void process() = 0;         // Node processor
                                            // comparing 2 Nodes
        virtual int compare(Node const &other) const = 0;

        virtual void rejected(Node const &twice)  // called when the node
        {                                   // was found in the tree
        }
};
```

### 18.2.2 The Tree class

The class `Tree` is responsible for the storage of objects which are derived from a `Node`. To implement the recursive tree structure, the class `Tree` has two `private` pointers as its data, pointing to subtrees: a `Tree *left` and `Tree *right`. The information which is contained in a node of the tree is represented as a `private` field `Node *info`.

`Tree` objects may be constructed empty and they may be constructed storing an initial `Node` object.

To scan a binary tree, the class `Tree` offers three methods: preorder, inorder and postorder. When scanning in preorder first the left subtree is scanned, then the leaf itself is processed and finally the right subtree is scanned. When scanning in inorder first a leaf in a node is processed, then the left subtree is scanned and finally the right subtree is scanned. When scanning in postorder first the left and right subtrees are scanned and then the leaf itself is processed.

The definition of the class `Tree` is given below:

```
#include "node.h"

class Tree
{
    public:
```

```
        // destructor, constructors
        ~Tree();
        Tree();
        Tree(Node const &node);
        Tree(Tree const &other);

        // assignment
        Tree const &operator=(Tree const &other);

        // addition of a Node
        void add(Node const &node);

        // processing order in the tree
        void preorder_walk();
        void inorder_walk();
        void postorder_walk();

    private:
        // primitives
        void construct(Tree const &other);
        void destroy();

        // called by add(Node const &node)
        void add(Tree *&branch, Node const &node);

        // data
        Tree
            *left,
            *right;
        Node
            *node;
};
```

### Constructing a tree

There are three constructors defined in the `Tree` class. The copy constructor is presented in the next section, the other two constructors are:

```
#include "tree.h"

// default constructor: initializes to 0
Tree::Tree()
:
    left(0),
    right(0),
    node(0)
{
}

// Node constructor: add a Node object
Tree::Tree(Node const &node)
```

```
    :
        left(0),
        right(0),
        node(node.clone())
    {
    }
```

### The 'standard' functions

As can be seen from the class definition, `Tree` contains pointer fields. This means that the class will need a destructor, a copy constructor and an overloaded assignment function to ensure that no allocation problems occur.

The destructor, the copy constructor and the overloaded assignment function are implemented with two primitive operations `construct()` and `destroy()` (presented later):

```
#include "tree.h"

// destructor: destroys the tree
Tree::~Tree()
{
    destroy();
}

// copy constructor: initializes to contents of other object
Tree::Tree(Tree const &other)
{
    construct(other);
}
```

### Adding an object to the tree

Adding a new object to the tree is a recursive process. When the function `add()` is called to insert an object into the tree, there are only three possibilities:

- The `node` field of the current node can be a 0-pointer. In that case, a clone of the `Node` object is inserted at the current node.
- When the tree is already partially filled, then it is necessary to determine whether the object to add should come 'before' or 'after' the object of the current node. This comparison is performed by `compare()`, a pure virtual function whose implementation is required by `Node`. Depending on the order the new object must be inserted in the `left` or in the `right` subtree. Adding a node to a subtree is done by an overloaded (private) `add()` function.
- When the comparison of the new object and the object of the current node yields 'equality', then the new object should not be stored again in the tree. The function `rejected()` is called to process the duplicated `Node`.

Here are the two `add()` functions:

```
#include "tree.h"

void Tree::add(Node const &newNode)
{
    if (!node)
    {
        node = newNode.clone();
        return;
    }

    int
        cmp = node->compare(newNode);

    if (!cmp)               // already stored
    {
        node->rejected(newNode);
        return;
    }

    add
    (
        cmp < 0 ? left : right, newNode
    );
}

void Tree::add(Tree *&tree, Node const &newNode)
{
    if (!tree)
        tree = new Tree(newNode);
    else
        tree->add(newNode);
}
```

#### Scanning the tree

The class `Tree` offers three methods of scanning a binary tree: preorder, inorder and postorder. The three functions defining these actions are recursive:

```
#include "tree.h"

void Tree::preorder_walk()
{
    if (left)
        left->preorder_walk();
    if (node)
        node->process();
    if (right)
        right->preorder_walk();
}
```

```
void Tree::inorder_walk()
{
    if (node)
        node->process();
    if (left)
        left->inorder_walk();
    if (right)
        right->inorder_walk();
}

void Tree::postorder_walk()
{
    if (left)
        left->postorder_walk();
    if (right)
        right->postorder_walk();
    if (node)
        node->process();
}
```

**The primitive operations copy() and destroy()**

The functions `copy()` and `destroy()` are two `private` member functions which implement primitive operations of the class `Tree`: the copying of the contents of another `Tree` or the destroying of the tree.

```
#include "tree.h"

void Tree::destroy()
{
    delete node;
    if (left)
        delete left;
    if (right)
        delete right;
}

void Tree::construct(Tree const &other)
{
    node = other.node ? other.node->clone() : 0;
    left = other.left ? new Tree(*other.left) : 0;
    right = other.right ? new Tree(*other.right) : 0;
}
```

Concerning this implementation we remark the following:

- The function `destroy()` is recursive, even though this is not at once visible. A statement like `delete left` will activate the destructor for the `Tree` object which is pointed to by `left`; this in turn will call `destroy()` etc..

- Similarly, the function `construct()` is recursive. The code `left = new Tree(*other.left)` activates the copy constructor, which in turn calls `construct()` for the left branch of the tree.
- As is the case with the function `add()`, nodes themselves are cloned by the function `clone()`. This function must be provided by a concrete implementation of a derived class of `Node`.

### 18.2.3 Using Tree and Node

We illustrate the usage of the classes `Tree` and `Node` by a program that counts words in files. Words are defined, rather blandly, as series of characters, separated by white spaces. The program shows which words are present in which file, and how many times.

Below is the listing of a class `Strnode`. This class is derived from `Node` and implements the virtual functions. Note how this class implements the counting of words; when a given word occurs more than one time, `Tree` will call the member function `rejected()`. This function simply increases the private counter variable `times`.

```
#include <fstream>
#include <iomanip>
#include <string>

#include "tree.h"

class Strnode: public Node
{
    public:
        Strnode(string const &s)
        :
            str(s),
            times(1)
        {
        }
        Node* clone() const
        {
            return (new Strnode(*this));
        }
        int compare (Node const &other) const
        {
            return
            (
                str.compare
                (
                    static_cast<Strnode const &>(other).str
                )
            );
        }
        void process ()
        {
            if (times)
                cout << setw(20) << str.c_str() << ": " <<
                        setw(3) << times << endl;
```

```
        }
        void rejected(Node const &node)
        {
            ++times;
        }
    private:
        string
            str;
        unsigned
            times;
};

void countfile(istream &inf)
{
    Tree
        tree;
    string
        word;

    while (inf >> word)
    {
        Strnode
            next(word);

        tree.add(next);
    }
    tree.inorder_walk();
}

int main (int argc, char **argv)
{
    int
        exitstatus = 0;

    if (argc > 1)
    {
        for (int i = 1; i < argc; ++i)
        {
            ifstream
                inf(argv[i]);

            if (!inf)
            {
                cerr << "can't open " << argv[i] << endl;
                exitstatus++;
                continue;
            }
            countfile(inf);
        }
    }
    else
        countfile (cin);
```

```
    return (exitstatus);
}
```

## 18.3 Classes to process program options

Programs usually can be given *options* by which the program can be configured to a particular task. Often programs have sensible default values for their options. Given those defaults, a *resource* file may be used to overrule the options that were hard-coded into the program. The resource file is normally used to configure the program to the specific needs of a particular computer system. Finally, the program can be given *command-line* options, by which the program can be configured to its task during one particular run.

In this section we will develop a set of classes starting from the class `Configuration`, whose objects can process a great variety of options. Actually, we'll start from a small demo program, in which an object of the class `Configuration` is used. From there, the class `Configuration` will be developed, working our way down to the auxiliary classes that are used with the `Configuration` class.

The resulting program will be available as a zip-file containing the sources and (Linux) binary program at our ftp-site. The zip-archive contains all the sources and auxiliary files for creating the program, as well as an icmake build script.

### 18.3.1 Functionality of the class Configuration

What functionality must a `Configuration` object have?

- Its *constructor* should get full control over the program arguments `int argc` and `char **argv`.
- The class will have several pointer data members. Consequently, the class will need a *destructor*.
- The `Configuration` object must be able to load a resourcefile. Our resource file will obey the standard *unix* form of configuration files: empty lines are ignored, and information on lines beyond the hashmark (#) is ignored.
- The `Configuration` object must be able to process command-line options, which can be either with or without an extra argument.
- The object should be able to produce the plain name of the program, i.e., the name from which all directories are stripped.
- The object should be able to produce the name of the resource file that was used.
- The object should be able to tell us how many command-line arguments are available, not counting command-line options and their arguments.
- The object should be able to produce the command-line arguments by their index-value, again not counting command-line options and their arguments.
- The object should be able to produce an option, given the name of the option. We don't know yet what an `Option` is, but then, we don't have to if we decide at this point that pointers to `Options`, rather than the `Options` themselves are prodcued.

Maybe of similar importance as the functionality the object *can* perform is what the object *can not* perform:

- A program will normally not need multiple `Configuration`objects. Therefore there will be *no copy constructor*.
- For the same reason, the class will have *no overloaded assignment operator*.

What if we accidently try to use a copy-constructor or (overloaded) assignment operator? Those situations will be covered by the following *trick*: we will *mention* a copy constructor and an overloaded assignment operator in the interface of the class, but will not implement it. The compiler will, where needed, happily generate code calling these two functions, but the program can't be linked, since the copy constructor and the overloaded assignment operator aren't available. Thus we prevent the accidental use of these functions. This approach is used also with other, auxiliary, classes.

Now that we've specified the functionality we're ready to take a look at the interface.

### The interface of the class Configuration

Here is the full interface of the `class Configuration`. In the interface, we recognize the functions we required when specifying the functionality of the class: the constructor, destructor, and the (not to be implemented) copy constructor and overloaded assignment operator.

To process the resource file we have `loadResourceFile()`, the command-line options are processed by `loadCommandLineOptions()`. Next we see two plain accessors: `programName()` will return the plain program name, while `resourceFile()` will return the name of the resource file. To obtain the number of command-line arguments that are available when all command-line options have been processed we have `argc()`. The arguments themselves are obtained by overloaded index operator, using an `unsigned` argument. Finally, options can be obtained by name: for this another overloaded index operator is available, this time using a string (`char const *`) for its argument.

The `private` section contains data: variables to access `argc` and `argv`, using reference-type variables; variables to store the program- and resource filenames, and two `Hashtables` (the `class Hashtable` will be covered in section 18.3.6) containing, respectively, the precompiled options and the command-line options.

Here is the interface of the class `Configuration`:

```
#ifndef _Configuration_H_
#define _Configuration_H_

#include "../hashtable/hashtable.h"

class Option;

class Configuration
{
    public:
        Configuration(int &argc, char const **&argv, int initialCap = 20,
                      double maxLoadFactor = 0.75);
```

```
        ~Configuration();

        Configuration(Configuration const &other);            // NI
        Configuration &operator=(Configuration const &right); // NI

        void loadResourceFile(char const *fname);
        void loadCommandLineOptions();
        char const *programName();      // name of the program
        char const *resourceFile();     // name of used resourcefile
        unsigned argc() const;          // count beyond [0], c.q. options
                                        // returns argv[index] | 0
                                        // also beyond [0] c.q. options
                                                        // option [name]
        Option const * operator[](char const *name) const;
        char const *operator[](unsigned index) const;   // argument[index]

    private:
        int
            argcShift,
            &argC;
        char const
            **&argv;
        char
            *progName;
        Hashtable
            optionTable,
            cmdLineOption;
        char
            *resourceFilename;
};

#include <string.h>
#include "../option/option.h"
#include "../string/string.h"
#include "../mem/mem.h"
#include "../ustream/ustream.h"
#include "../stringtokenizer/stringtokenizer.h"

#endif  _Configuration_H_
```

### An example of a program using the class Configuration

Below we present the source of the demonstration program. The program sets up the memoryhandler, to make sure that failing memory allocations will be noticed.

Next, a configuration object is created. This object is passed to an auxiliary function showing us interesting aspects of the object (`showConfigurationInformation()`). Although this function tells us things about the `Configuration` object, it was not made part of the class, since it was specifically designed in the context of the demonstration program, without adding any real functionality to the `Configuration` class.

Having displayed the raw information stored in the `Configuration` object, the resource-file is loaded. This might alter the values of the program-parameters, of which there are four in the demonstration program. Having loaded the resourcefile, the contents of the `Configuration` object are shown again.

Then, the command-line options (if any) are processed, followed by yet another display of the contents of the `Configuration` object.

Here is the source of the demonstration program:

```
#include "demo.h"

int main(int argc, char const **argv)
{
    Mem::installNewHandler();

    Configuration
        config(argc, argv);

    showConfigurationInformation(config, "After constructing 'config'");

    config.loadResourceFile("demo.rc");

    showConfigurationInformation(config, "After reading demo.rc");

    config.loadCommandLineOptions();

    showConfigurationInformation(config,
            "After processing command-line options");

    return (0);
}
```

### 18.3.2 Implementation of the class Configuration

#### The constructor

The constructor of the class `Configuration` expects `argc` and `argv` as reference-type variables. Apart from these two, tho extra parameters are defined, for which the interface defines default values: `initialCap` defines the initial capacity of the hashtables that are used by the `Configuration` object, and `maxLoadFactor` defining the maximum load percentage of the hashtables. So, with the default parameters the hashtables would be enlarged once more than 15 elements are stored in them.

Having initialized the reference variables and the hashtables the options are stored in the hashtables for fast access. The `Option`-class function `nextOptionDefinition()` produces a sequence of all options that are defined for the program. Each option's name and value is stored in the `optionTable` hashtable, and each option's command-line character and name is stored in the `cmdLineOption` hashtable. Therefore, the values of options can be retrieved immediately, given the name of the option, while the option's command-line character can be used to produce the name of the option, which can then be used in a second step to obtain the value of the option.

Here is the source of the constructor:

```
#include "configuration.h"

Configuration::Configuration(int &argCount, char const **&argVector,
                             int initialCap, double maxLoadFactor)
:
    argC(argCount),
    argv(argVector),
    optionTable(initialCap, maxLoadFactor),
    cmdLineOption(initialCap, maxLoadFactor)
{
    resourceFilename = Mem::strdup("");

    Option
        *option;

    while ((option = Option::nextOptionDefinition()))
    {
        String
            *name = new String(option->getName());

        optionTable.put(name, option);

        String const
            *cmdopt =  &(option->getCmdLineOption());

        if (strlen(*cmdopt))
            cmdLineOption.put(new String(*cmdopt), new String(*name));
    }

    char const
        *cp = strrchr(argv[0], '/');

    progName =
        Mem::strdup
        (
            !cp ?
                argv[0]
            :
                cp + 1
        );

    argcShift = 1;
}
```

### loadResourceFile()

The function `loadResourceFile()` processes a *unix*-style resource-files. In these files, empty lines are ignored, as well as information on a line beyond hash-marks (`#`) *if* these hashmarks are preceded by the beginning of the line or white space. Long lines may be stretched out over several lines by adding a continuation character (the backslash (\)) at the end of each line that continues on the next line.

To obtain the remaining lines of the configuration file, `loadResourceFile()` creates a `Ustream` object. The class `Ustream` was specifically designed for the processing of *unix*-style resource-files. As this class doesn't add much to the understanding of the `Configuration`-class its interface and implementation is not discussed in the annotations. Rather, interface and implementation is found in the configdemo.zip file at our ftp-site.

The processing of the information in the configuration file is based on the assumption that all information on a line is organized as follows:

- The first word is an identifying word: it should match the name of an option. The word is called the *key*.
- The key is optionally terminated by a colon, e.g.,

  ```
  color:
  ```

- The remainder of the line, starting at the first non-blank character beyond the key, and ending at the last non-blank character on the line, is considered to be the *value* of the key.

With respect to this format, each key is looked up in the `optionTable`. If found, the value of the option is set to the key's value. Otherwise, if the key is not found, a warning message is written, by catching the exception thrown by the hashtable when it receives an undefined option-name.

Apart from the `Ustream` object, the function `loadResourceFile()` also uses a `StringTokenizer` object, which splits lines from the `Ustream` file into words. The first word is interpreted as `key`, while the function `range(index)` produces the unsplit line beyond word `index`. The class `StringTokenizer` is also found in the distributed `zip`-file.

### loadCommandLineOptions()

The function `loadCommandLineOptions()` uses the function `getopt()` which is available on *unix* systems to retrieve command-line options (and possibly their values) and to separate them from the remaining command-line arguments. The function `getopt()` expects (among other arguments) a string of command-line option letters, which are possibly followed by a colon. If a colon is following a command-line option, then information trailing the command-line option character or the next command-line argument is interpreted as the *value* of the command-line option. E.g., a command-line option character specified as `n:` may be specified on the command-line as `-n20` or `-n 20`.

The function `Hashtable::catKeys()` is used to obtain a list of command-line option characters. Next, the options are extracted from the command-line arguments using `getopt()`. When an option has been found, the `cmdLineOption` hashtable is used to obtain the name of the option, then the `optionTable` hashtable is used to obtain a pointer to the option.

Next the option receives a new value, through the virtual function `assign()`. This function is available for all options, and allows `loadCommandLineOptions()` to assign a new value to an option irrespective of the actual type of the option.

Here is the code of the function `loadCommandLineOptions()`:

```
#include "configuration.h"

void  Configuration::loadCommandLineOptions()
{
    String
        list;

    cmdLineOption.catKeys(list);

    register int
        optionChar;
    String
        opt;
    register char
        *cp;
    opterr = 0;                         // no error messages from getopt()
    while                               // while options are found
    (
        (optionChar = getopt(argC, (char *const *)argv, list)) != -1
        &&
        (cp = strchr(list, optionChar))
    )
    {
        opt = " :";

        opt[0] = (char)optionChar;      // create option-string
        if (cp[1] != ':')               // no option value ?
            opt[1] = 0;                 // then remove ':' from opt.

        Option                          // get the configuration option
            *option = (Option *)optionTable[cmdLineOption[&opt]];

        option->assign(optarg);         // assign the value
    }
    argcShift = optind;                 // first non-option index in argv
}
```

### 18.3.3 The class Option

The class `Option` is designed as an abstract base class, defining the protocol to which all derived classes must adhere. Derived classes representing logical values (`Boolean`), integer values (`Int`), real values (`Double`) and textstrings (`Text`) will be constructed later on.

The class itself is derived from another abstract base class, `Object`. Pointers to `Object`s are stored in, e.g., `Hashtable`s.

The class `Option` (cf. section 18.3.3), has a constructor, expecting an option *name* and the specification of a command-line parameter, and a *virtual* destructor to be able to deleting memory allocated by derived class objects through an `Option` pointer.

Default implementations returning the logical, int, double and textvalues of options are available as well. These implementations are replaced in derived classes by member functions returning the real, rather than the default, value of the derived class' object.

Since the options must be storable in a hashtable, and since the hashtable must be able to compare two different object for equality, abstract members `hashCode()` and `equals()` are available, to be implemented in the derived class' objects.

The name and command-line option are obtained via two accessor functions: `getName()` and `getCmdLineOption()`, respectively.

To assign a value to an option one more function must be implemented by derived class options: `assign()`, to assign a value to an option.

The `static Option *nextOptionDefinition()` member function returns a pointer to an object of a class derived from `Option`. The returned option is constructed by a function that can be called from an element of the

```
static Option *(*optionConstructor[])(Mold const &mold)
```

array of pointers to functions returning pointers to `Option`s. Each of these functions expects a reference to a `Mold` struct.

An array of these structs must be available as `static Mold mold[]`. The `Mold` array allows us to specify as *data* the ingredients of any option we require in our program. In other words: by defining the elements of an array `Option::Mold Option::mold[]` all kinds of program-options and their default values. can easily be defined.

For example, in our demonstration program four program options were defined, representing a logical value, an integer value, a real value and a textual string. Note that the following `mold[]` array is defined as *data*:

```
#include "../demo.h"

Option::Mold Option::mold[] =
{
    {Boolean,   "colors",   "c",    "True"},
    {Int,       "trials",   "n:",   "20"},
    {Double,    "epsilon",  "e:",   "0.004"},
    {Text,      "files",    0,      "ls -Fla"},
    {},
};
```

The last element of the `mold[]` array is an empty `struct`, acting as a sentinel. The remaining lines (refer to the struct Mold in the interface of the class `Option`) contain four elements:

- The first element indicates the type of option: the options mentioned in the `Type` enum are available. Note that this `enum` is *protected*: it's only used in derived classes.

- The second element is the name of the option, as it should appear in resource files and in the `Configuration`'s overloaded index operator.
- The third element is the command-line option character. If set to zero, there is no command-line option. If the command-line option is followed by a colon, then the command-line option should be given an argument of its own.
- The fourth element is the initial default value of the option. For logical (`Boolean`) options string values like `on, off, true, false, 0, 1` in any casing are all acceptable. Note again that the initial default values are given as strings.

### The interface of the class Option

Here is the complete interface of the abstract base class `Option`:

```
#ifndef _Option_H_
#define _Option_H_

#include "../string/string.h"

class Option: public Object
{
    public:
        Option(char const *name, char const *cmdLineOpt);
        ~Option();

        virtual int         BoolValue()     const;
        virtual int         IntValue()      const;
        virtual double      DoubleValue()   const;
        virtual char const *TextValue()     const;

        unsigned    hashCode()                      const;
        int         operator==(Object const &other) const;

        String const
            &getName() const,
            &getCmdLineOption() const;

        virtual void assign(char const *string) = 0;

        static Option *nextOptionDefinition();
    protected:
        enum Type
        {
            Sentinel,
            Int,
            Double,
            Text,
            Boolean,
        };

    private:
```

```
        struct Mold
        {
            Type
                optionType;
            char
                *name,
                *cmdLineOption,
                *defaultValue;
        };

        static Mold
            mold[];

        static Option *(*optionConstructor[])(Mold const &mold);

        String
            name,
            cmdLineName;
};

#include <strstream.h>
#include "../booloption/booloption.h"
#include "../intoption/intoption.h"
#include "../doubleoption/doubleoption.h"
#include "../textoption/textoption.h"

#endif  _Option_H_
```

### The static member nextOptionDefinition

The static member function `nextOptionDefinition()` is called repeatedly until it returns 0. The function visits all elements of the `mold[]` array, calling the static function `optionConstructor` associated with the option-type of the element of the array `mold[]` that is visited.

The variable `optionConstructor[]` is an array, which is initialized as data of the `class Option`. The elements of the `optionConstructor[]` array are pointers to `Constructor()` functions of all the derived classes. These functions construct actual derived class option objects, and expect the ingredients for the construction as a reference to a `Mold struct`.

The function `nextOptionDefinition()` is:

```
#include "option.h"

Option *Option::nextOptionDefinition()
{
    static unsigned
        index = 0;

    if (mold[index].optionType == Sentinel)
        return (0);
```

```
    Option
        *option =
            optionConstructor[mold[index].optionType]
            (mold[index]);

    index++;
    return (option);
}
```

The array optionConstructor[] is initialized as follows:

```
#include "option.h"

Option *(*Option::optionConstructor[])(Mold const &mold) =
{
    0,
    IntOption::Constructor,
    DoubleOption::Constructor,
    TextOption::Constructor,
    BoolOption::Constructor,
};
```

Note that in this initialization reflects the ordering of the Option::Type enum. There is no constructor for the Sentinel enum-value, while the remaining elements contain the addresses for the different derived-class option types.

### 18.3.4 Derived from Option: The class TextOption

Below (in section 18.3.4) the interface of the class TextOption, derived from Option, is given. The class contains implementations of all the pure virtual functions of the class Option, and it mentions the existence of a copy constructor and overloaded assignment operator. However, these functions are (once again) not to be used, and are mentioned here as a safeguard against their being used accidently.

The interesting part of the interface is the function static Option *Constructor(Mold const &mold): it constructs a TextOption object (through TextOption's constructor), using the ingredients it encounters in the Mold it receives as its argument. Note that the prototype of Constructor corresponds to the prototype of the elements of the array Option::optionConstructor[]. As we have seen (in section 18.3.3), Option:optionConstructor[Text] has been given the value TextOption::Constructor, thus setting up the connection between an option-type and the constructor for such an option from the ingredients found in an Option::Mold.

The other three classes derived from the class Option are constructed similarly. The reader is referred to their interfaces and implementation in the zip-archive in our ftp-site.

#### The interface of the class TextOption

Here is the interface of the class TextOption, derived from Option:

```
#ifndef _TextOption_H_
#define _TextOption_H_

#include "../option/option.h"

class TextOption: public Option
{
    public:
        static Option *Constructor(Mold const &mold);
        TextOption(char const *name, char const *cmdLineOpt,
                   char const *initialValue);
        ~TextOption();

        TextOption(TextOption const &other);                // NI
        TextOption &operator=(TextOption const &other);     // NI

        void assign(char const *str);
        char const *TextValue() const;
        char const *toString() const;
    private:
        char
            *value;
};

#include "../mem/mem.h"

#endif  _TextOption_H_
```

### The implementation of the assign() function

As an example of an implementation of an `assign()` function, we present the function `TextOption::assign()`. As defined by the interface of the class Option, this function has one parameter, a `char const *str`. It needs to perform only two tasks: First, the old value of the `TextOption` object is deleted, then a new value is assigned. Corresponding `assign()` functions are available for the other derived option classes.

Here is the implementation of `TextOption::assign()`:

```
#include "textoption.h"

void TextOption::assign(char const *str)
{
    delete value;
    value = Mem::strdup(str);
}
```

### 18.3.5 The class Object

The `class Object` is an abstract base class. Pointers to `Objects` are be stored in `Hashtables`. The class is a very simple one, containing a virtual destructor (doing nothing in particular), and requiring the implementation of three pure virtual functions:

- `int operator==(Object const &other)`, used to compare two objects of classes derived from the class `Object`,
- `unsigned hashCode()`, returning a hashcode for the object. This function is used in combination with a `Hashtable` object.
- `char const *toString()`, returning a printable representation of the object.

Here is the interface of the class `Object`:

```
#ifndef _Object_H_
#define _Object_H_

class Object
{
    public:
        virtual ~Object();

        virtual int         operator==(Object const &other) const = 0;
        virtual unsigned    hashCode()                   const = 0;
        virtual char const *toString()                   const = 0;
};

#endif  _Object_H_
```

### 18.3.6 The class Hashtable

The `class Hashtable` is used to store and retrieve objects of classes derived from the class `Object`. The class contains two pointers to vectors of pointers to `Objects`, containing the *keys* and *values* that are stored in the hashtable. Furthermore, the class has data-members holding the actual number of elements that are stored in the hashtable (`n`), the number of elements of the two vectors of pointers to `Objects` (`capacity`), the original number of elements of these vectors (`initialCapacity`) and the maximum proportion of elements of the vectors that may be occupied (`maxLoadFactor`).

The `Hashtable` objects are self-expanding. Once `maxLoadFactor` threatens to be exceeded, the table is expanded automatically.

The functionality of the hashtable includes members for retrieving values of the objects stored in the table using either the name of a key (as a `char const *`) or a pointer to an `Object`; a member to add a new key/value pair to the table, and a utility member `catKeys()` returning a string containing the catenated names of all keys. This latter function is used by the `Option::nextOptionDefinition()` to tell `getopt()` what command-line option characters it can expect.

The interface of the `class Hashtable` also shows some private member functions, used for expanding the table, and for inserting and retrieving elements from the table. Some of these functions are covered in the following discussion. Functions not needing special attention are available in the zip-archive.

Here is the interface of the `class Hashtable`:

```
#ifndef _Hashtable_H_
#define _Hashtable_H_

#include "../string/string.h"

class Object;

class Hashtable
{
    public:
        Hashtable(int initialCapacity, double maxLoadFactor = 0.75);
        ~Hashtable();

        Hashtable(Hashtable const &other);                  // NI
        Hashtable const &operator=(Hashtable const &other); // NI

        Object const *operator[](Object const *key) const;
        Object const *operator[](char const *key) const;
        Object const *put(Object *key, Object *value);  // returns value

        void catKeys(String &target);               // catenate the keys
                                                    // as strings
        char const *toString() const;
    private:
        void installVectors(int capacityRequest);
        int lookup(Object const *key) const;    // key must exist
        int mayInsert(Object *key);             // key might not exist

                                            // the key in the table
        int expanded();                     // 1 if table was expanded

        unsigned
            capacity,
            initialCapacity,
            n;
        double
            maxLoadFactor;
        Object
            **keys,
            **values;
};

#include <unistd.h>
#include <stdlib.h>

#include "../option/option.h"
```

```
#endif  _Hashtable_H_
```

### The Hashtable constructor

The constructor of the hashtable initializes the data-members of the table, and then calls `installVectors()` to initialize the `keys` and `values` vectors. Here is the constructor of the `class Hashtable`:

```
#include "hashtable.h"

Hashtable::Hashtable(int iniCap, double maxFactor)
{
    maxLoadFactor = maxFactor;
    n = 0;
    initialCapacity = iniCap;

    capacity = 0;
    keys = 0;
    values = 0;

    installVectors(initialCapacity);
}
```

The function `installVectors()` simply creates two vectors of the required number of elements (i.e., `capacity`), initializing the vectors with null-pointers.

### The function mayInsert()

The functions `mayInsert()` returns the index of a key that is stored in the hashtable. The difference with the function `lookup()` is that the function `lookup()` requires the key to be available in the hashtable, whereas the function `mayInsert()` will insert the key when it isn't available yet.

If the function `lookup()` doesn't find the key in the table, it throws a `char const *` exeption, containing the name of the key. The exception is thereupon caught by the function `Configuration::loadResourceFile()`. The function `mayInsert()`, however, will try to insert a non-existing key into the hashtable.

Before looking for a key, both `lookup()` and `mayInsert()` first determine an initial hashcode, using the key's `hashCode()` function. A simple add-the-hash rehash scheme is used to cope with collisions. The add-the-hash value is at least 1 and at most the current capacity minus one. Using a prime-sized hashtable, this ensures that all elements of the hashtable are visited by repeatedly adding the add-the-hash value to the index value that was last used.

The insertion process itself consists of a perpetual loop, that terminates when the index of the key in the hashtable has been determined.

If an empty element of the `key` vector is hit, `expand()` is called, which may enlarge the hashtable. If the table was enlarged, both the hashcode and the add-the-hash value of the actual key are recomputed, and the perpetual loop starts its next cycle. Otherwise, the key is entered at the empty element's position, and its index value is returned.

If the key is found in the vector of keys, then the corresponding index position is returned. Alternatively, a collision may occur, and the index value is incremented by the add-the-hash value, followed by the next cycle of the perpetual loop.

Thus, the `lookup()` and `mayInsert()` functions return the index of the provided key. Apart from that, `lookup()` will throw an exception when the provided key isn't found in the table.

Here is the sourcetext of the function `mayInsert()`:

```
#include "hashtable.h"

//  addTheHash is set in the range 1 .. capacity - 1, and the initial
//  index is made equal to the addTheHash value. Since addTheHash is non-zero
//  a new index computed by adding the addTheHash value to the index will
//  always get another value. The zeroth index of the hashtable will only be
//  used as the result of a collision, but that doesn't matter: hashtables
//  aren't filled up completely anyway.

int Hashtable::mayInsert(Object *key)
{
    unsigned
        hashCode = key->hashCode();
    register unsigned
        addTheHash = 1 + hashCode % (capacity - 1),
        index = addTheHash;                 // within the capacity range

    while (1)
    {
        if (!keys[index])                   // empty slot ?
        {
            if (expanded())                 // hashtable was expanded ?
            {
                addTheHash = 1 + hashCode % (capacity - 1);
                index = addTheHash;         // new index after expansion

                continue;                   // restart the checking
            }
            keys[index] = key;              // place the key here
            ++n;                            // n contains #-elements

            return (index);                 // and produce its index
        }

        if (*keys[index] == *key)       // same object ?
            return (index);                 // return its index

        if ((index += addTheHash) >= capacity)   // collision: try next entry
            index -= capacity;
    }
}
```

### The function expanded()

The function `expanded()` first checks the loadfactor of the hashtable: if the actual number of elements divided by the capacity of the table exceeds `maxLoadFactor`, the current `keys` and `values` vectors are saved, and new vectors containing `initialCapacity` extra elements are installed.

Next, the elements of the old `keys` vector are visited. If a non-empty element is found, that element and its value are stored in the hashtable using the function `put()`. This process continues until `n` elements (the number of non-empty elements in the old vectors) are stored in the enlarged table. Since the function `put()` *owns* the objects that its arguments point to (i.e., `Object` $*$s rather than `Object const` $*$s are used, the objects the elements of the old vectors point to must *not* be deleted. Therefore, at the end of the function `expanded()` the old keys and values vectors are simply deleted, disregarding the objects their elements point to.

## 18.3.7 Auxiliary classes

The classes we've covered so far rely on the specific functionality of other classes. The memory management class `Mem` is a good example: while standard functions are available for the allocation of memory, these functions reduce to the function `malloc()`, and not to the operator `new`. Since the operator `new` can be protected by the `set_new_handler()` function, it's a good idea to duplicate the popular standard memory allocating functions based on `malloc()` by functions using `new`.

Another example is found in the class `Util`, containing functions we think are useful, but which we could not place conceptually easy in other classes. For example, the utility class contains a function `prime()` returning a prime number.

The following utility classes are available:

- `Mem`: this class handles memory allocation through the operator `new` rather than through the function `malloc()`.
- `String`: objects of this class represent strings, and can perform certain string-related tasks.
- `StringTokenizer`: objects of this class break up strings into substrings according to a set of delimiters.
- `Ustream`: objects of this class handle *unix*-style configuration files, in which empty lines and information on lines beyond the hash-mark are ignored.
- `Util`: this class contains functions performing tasks which do not belong conceptually to other classes.

The `Mem` and `Util` classes contain just static member functions, and do not require objects to be used. For the other classes objects must be defined.

The next sections will cover the interfaces of these classes. The implementation of the functions of these classes is found in the zip-archive at our ftp-site.

### The class Mem

The class `Mem` contains functions related to the allocation of memory, using the operator `new`. Using `new`, it is easy to catch exhausted dynamic memory through the function `set_new_handler()`.

The class contains functions to install a `new`-handler, to duplicate and concatenate strings, to compare strings, and to reallocate memory. As all these functions are static, there is no need to create a `Mem` object.

The function `realloc()` isn't a particularly elegant attempt to make available a function that resembles the standard `malloc()`-based `realloc()` function. Actually, in the demonstration program it's used only by the `StringTokenizer` constructor. However, by making it a member of the latter class, we feel we would mix up memory allocation with string handling.

The `Mem::realloc()` function does a rather crude job: it should be used only for enlarging the required amount of memory, in which case the extra allocated memory remains completely uninitialized.

The other member functions are implemented in a standard way. Most of them accept null-pointers as arguments as well. Here is the interface of the `class Mem`:

```
#ifndef _Mem_H_
#define _Mem_H_

class Mem
{
    public:
        static void installNewHandler();
        static char *strdup(char const *str);
        static int casecmp(char const *s1, char const *s2);
        static int cmp(char const *s1, char const *s2);
        static char *strndup(char const *str, unsigned len);
        static char *strcat(char const *src1, char const *src2);
        static void *realloc(void *addressOfPointerToOldData,
                            unsigned dataSize, unsigned oldN,
                            unsigned newN);
    private:
        static void memoryExhausted();
};

#include <iostream.h>
#include <new.h>
#include <string.h>

#endif  _Mem_H_
```

### The class String

Objects of the class `String` represent strings: 0-delimited series of ascii-characters. The class is derived from `Object`, so `String` objects can be stored in `Hashtables`.

Apart from the functions required by the class `Object`, the class `String` contains all standard members, like a copy constructor and a overloaded assignment operators. Apart from these members, there is a conversion operator, allowing the use of a `String` object as a `char const *`, and there are members for enlarging the string by catenating another string to it, and for retrieving a character using the index-operator.

Here is the interface of the class `String`:

```
#ifndef _String_H_
#define _String_H_

#include <iostream.h>
#include <stdarg.h>

#include "../object/object.h"

class String: public Object
{
    public:
        String();
        String(char const *arg);
        ~String();

        String(String const &other);
        String &operator=(String const &rvalue);
        String &operator=(char const *rvalue);

        int operator==(Object const &other) const;
        unsigned hashCode() const;
        char const *toString() const;

        operator char const *() const;
        String &strcat(char const *str2);
        char &operator[](unsigned index);
    private:
        char
            *string;
};

#include "../mem/mem.h"
#include "../hashtable/hashtable.h"

#endif  _String_H_
```

### The class StringTokenizer

The class `StringTokenizer` is used for breaking up strings into substrings according to a (set of) delimiters. By default, the white-space delimiters are used. The constructor of the class expects an ascii-z string (and optionally a string of delimiter-characters) and will split the string into substrings according to the set of delimiters.

The substrings are retrievable through the overloaded index-operator, returning pointers to `String` objects, which are then owned by the calling function. Another member function is `range()`, returning the substring starting at a particular index-position. For example, if `StringTokenizer st` contains five substrings, `st.range(3)` will return the substring of the original string starting at `st[3]`.

Here is the interface of the class `StringTokenizer`:

```
#ifndef _StringTokenizer_H_
#define _StringTokenizer_H_

#include "../string/string.h"

class StringTokenizer
{
    public:
        StringTokenizer(char const *cp, char const *delimiters = " \t\n");
        ~StringTokenizer();

        StringTokenizer(StringTokenizer const &other);              // NI
        StringTokenizer &operator=(StringTokenizer const &other);   // NI

        String *operator[](unsigned index);
        String *range(unsigned from);       // until the last one

    private:
        struct SubString
        {
            char
                *str;
            unsigned
                length;
        };

        char
            *str;
        SubString
            *subString;
        unsigned
            n;
};

#endif  _StringTokenizer_H_
```

### The class Ustream

The class `Ustream` processes files as *unix*-like configuration files. In these files empty lines are ignored, as is information starting at a hash-mark at the beginning of a line or preceded by a white-space character. Furthermore, lines are combined if the last character of a line is a backslash.

The constructor of the class expects one argument: the name of the file to be processed. Having created a `Ustream` object, the conversion operator `operator void *()` can be used to determine the successful opening of the file: it returns `0` if the file wasn't opened successfully.

The (non-empty, non-comment section of) lines of the file are returned by the member `read()` as a `char *`: the line is owned by the calling function. Calling `read()` succeeds until a null-pointer is returned.

After a successful read-operation, the member-function `lineNr()` will return the actual linenumber of the just read line in the original file. In this case empty and comment-lines *are* counted.

The file is closed when the `Ustream` object is destroyed.

Here is the interface of the class `Ustream`:

```
#ifndef _Ustream_H_
#define _Ustream_H_

#include <fstream.h>

class Ustream
{
    public:
        Ustream(char const *fname);

        Ustream(Ustream const &other);                  // NI
        Ustream const &operator=(Ustream const &right); // NI

        operator void *();              // direct status-check

        char *read();                   // 0 if no more lines
        int lineNr();

    private:
        ifstream
            stream;
        int
            line;
};

#include "../mem/mem.h"
```

```
#endif  _Ustream_H_
```

### The class Util

The class `Util` contains several utility functions, which did not belong elsewhere. The functions `atod()` and `atoi()` convert, respectively, strings to doubles and strings to ints, and they differ from the standard functions `atof()` and `atoi()` only by the fact that the `Util` functions accept null-pointers as well.

The function `prime()` uses the sieve of Aristosthenes to generate the first prime exceeding the value given as its argument.

The function `hashPjw()` returns a hashvalue for a string. This algorithm is given in *Aho, Sethi, and Ullman*'s *Compilers: Principles, Techniques and Tools, 1986, p. 435* as *P. J. Weinberger's* algorithm for computing hash-values.

The interface of the class `Util` is given below:

```
#ifndef _Util_H_
#define _Util_H_

#include <values.h>
    // uses INTBITS to find the # of bits in a word, hence in an int

class Util
{
    public:
        static double atod(char const *value);      // convert to double
        static int atoi(char const *value);         // convert to int
        static unsigned prime(unsigned lowerBound); // first prime exceeding
                                                    // lowerBound
        static unsigned hashPjw(char const *key);   // return hashvalue
    private:
        int const
            bitsPerInt = INTBITS,
            moduloMask = bitsPerInt - 1;
        static int
            shiftBitsPerInt;
};

#include <stdlib.h>
#include <string.h>
#include <math.h>

#endif  _Util_H_
```

## 18.4 Using Bison and Flex

The example discussed in this section digs into the peculiarities of using a parser- and scanner-generator with **C++**. Once the input for a program exceeds a certain level of complexity, it's advantageous to use a scanner- and parser-generator for creating the code which does the actual input recognition. The example about this topic assumes that the reader knows how to use the scanner generator `flex` and the parser generator `bison`. Both `bison` and `flex` are well documented elsewhere. The original predecessors of `bison` and `flex`, called `yacc` and `lex` are described in several books, e.g. in O'Reilly's book 'lex & yacc'.

However, the scanner and parser generators are also (and maybe even more commonly, nowadays) available as free software. Both `bison` and `flex` can be obtained from prep.ai.mit.edu/pub/gnu. `Flex` will create a `C++` class when called as `flex++`, or when the `-+` flag is used. With `bison` the situation is a bit more complex. Scattered over the Internet several `bison++` archives can be found (e.g., in rzbsdi01.uni-trier.de). The information in these archives usually dates back to 1993, irrespective of the version number mentioned with the archive itself. (However, the given ftp-archive also contains dos-executables, for those who are interested....)

Using `flex++` and `bison++` a `class`-based scanner and parser can be generated. The advantage of this approach is that the interface to the scanner and the parser tends to become a bit cleaner than without using the `class` interface.

Below two examples are given. In the first example only a lexical scanner is used to monitor the production of a file from several parts. This example focuses on the lexical scanner, and on switching files while churning through the parts. The second example uses both a scanner and a parser to transform standard arithmetic expressions to their postfix notation, commonly encountered in code generated by compilers and in `HP`-calculators. The second example focuses on the parser.

### 18.4.1 Using Flex++ to create a scanner

In this example a lexical scanner is used to monitor the production of a file from several parts. This example focuses on the lexical scanner, and on switching files while churning through the parts. The setup is as follows: The input-language knows of an `#include` statement, which is followed by a string indicating the file which should be included at the location of the `#include`.

In order to avoid complexities that have nothing to do with the current example, the format of the `#include` statement is restricted to the form `#include` <`filepath`>. The file specified between the pointed brackets should be available at the location indicated by `filepath`. If the file is not available, the program should terminate using a proper error message.

The program is started with one or two filename arguments. If the program is started with just one filename argument, the output is written to the standard output stream `cout`. Otherwise, the output is written to the stream whose name is given as the program's second argument.

The program uses a maximum nesting depth. Once the maximum is exceeded, the program terminates with an appropriate error message. In that case, the filenamestack indicating where which file was included should be printed.

One minor extra feature is that comment-lines should be recognized: `include` directives in comment-lines should be ignored, comment being the standard **C++** comment-types.

The program is created in the following steps:

- First, the file `lexer` is constructed, containing the specifications of the input-language.
- From the specifications in `lexer` the requirements for the `class Scanner` evolve. The `Scanner` class is a wrapper around the class `yyFlexLexer` generated by `flex++`. The requirements results in the specification of the interface for the class `Scanner`.
- Next, the `main()` function is constructed. A `Startup` object is created to inspect the commandline arguments. If successful, the `scanner`'s member `yylex()` is called to construct the output file.
- Now that the global setup of the program has been specified, the member functions of the different classes are constructed.
- Finally, the program is compiled and linked.

#### The flex++ specification file

The organization of the lexical scanner specification file is similar to the one used with `flex`. However, `flex++` creates a class (`yyFlexLexer`) from which the `class Scanner` will be derived.

The code associated with the regular expression rules will be located inside the class `yyFlexLexer`. However, it would be handy to access the member-functions of the derived class within that code. Fortunately, class derivation and inheritance helps us to realize this. In the specification of the class `yyFlexLexer()`, we notice that the function `yylex()` is a *virtual* function. In the `FlexLexer.h` header file we see `virtual int yylex()`:

```
class yyFlexLexer: public FlexLexer
{
    public:
        yyFlexLexer( istream* arg_yyin = 0, ostream* arg_yyout = 0 );

        virtual ~yyFlexLexer();

        void yy_switch_to_buffer( struct yy_buffer_state* new_buffer );
        struct yy_buffer_state* yy_create_buffer( istream* s, int size );
        void yy_delete_buffer( struct yy_buffer_state* b );
        void yyrestart( istream* s );

        virtual int yylex();
        virtual void switch_streams( istream* new_in, ostream* new_out );

    protected:
        ...
};
```

Consequently, if `yylex()` is defined in a derived class, then this function of the derived class will be called from a base class (i.e., `yyFlexLexer`) pointer. Since the `yylex()` function of the derived class is called, that function will have access to the members of its class, and to the public and protected members of its base class.

The context in which the generated scanner is placed is (by default) the function `yyFlexLexer::yylex()`. However, this context can be changed by defining the `YY_DECL`-macro. This macro, if defined, determines the context in which the generated scanner will be placed. So, in order to make the generated scanner part of the *derived class* function `yylex()`, three things must be done:

- The macro `YY_DECL` must be defined in the lexer specficiation file. It must define the derived class function `yylex()` as the scanner function. For example:

  ```
  #define YY_DECL int Scanner::yylex()
  ```

- The function `yylex()` must be declared in the class definition of the derived class.
- As the function `yyFlexLexer::yylex()` is a *virtual* function, it must still be defined. It is not called, though, so its definition may be a simple

```
int yyFlexLexer::yylex()
{
    return (0);
}
```

The definition of the `YY_DECL` macro and the `yyFlexLexer::yylex()` function can conveniently be placed in the lexer specification file, as shown below.

Looking at the regular expressions themselves, notice that we'll need rules for the recognition of the comment, for the `include` directive, and for the remaining characters. This is all fairly standard practice. When an include directive is detected, the derived-class' member function `switchSource()` is called, which will perform the required file switching. When the end of the file (`EOF`) is detected, the derived class' member function `popSource()` is called, which will pop the previous previously pushed file, returning 1. Once the file-stack is empty, the function will return 0, resulting in the call of `yyterminate()`, which will terminate the scanner.

The lexical scanner specification file has three sections: a **C++** *preamble*, containing code which can be used in the code defining the actions to be performed once a regular expression is matched, a `Flex++ symbol area`, which is used for the definition of symbols, like a mini scanner, or `options`, like `%option yylineno` when the lexical scanner should keep track of the line numbers of the files it is scanning and, finally a *rules* section, in which the regular expressions and their actions are given. In the current example, the lexer should mainly copy information from the `istream *yyin` to the `ostream *yyout`, for which the predefined macro `ECHO` can be used.

Here is the complete and annotated lexical scanner specification file to be used with `flex++`:

```
%{
/* ----------------------------------------------------------------------------
                                C++ -preamble.
   Include header files, other than those generated by flex++ and bison++.
      E.g., include the interface to the class derived from yyFlexLexer
----------------------------------------------------------------------------*/

                            // the yylex() function that's actually
                            // used
#define YY_DECL int Scanner::yylex()
```

```
#include "scanner.h"        // The interface of the derived class

int yyFlexLexer::yylex()    // not called: overruled by
{                           // Scanner::yylex()
    return (0);
}

%}

/* ----------------------------------------------------------------------------
                              Flex++ symbol area
                              ~~~~~~~~~~~~~~~~~~
      The symbols mentioned here are used for defining e.g., a miniscanner
---------------------------------------------------------------------------- */
%x comment
%option yylineno

eolnComment     "//".*
anyChar         .|\n

/* ----------------------------------------------------------------------------
                               Flex rules area:
                               ~~~~~~~~~~~~~~~~
     Regular expressions below here define what the lexer will recognize.
---------------------------------------------------------------------------- */
%%
    /*
        The comment-rules: comment lines are ignored.
    */
{eolnComment}
"/*"                    BEGIN comment;
<comment>{anyChar}
<comment>"*/"           BEGIN INITIAL;

    /*
        File switching: #include <filepath>
    */
"#include "[^>]*">"     switchSource();

    /*
        The default rules: eating all the rest, echoing it to output
    */
{anyChar}               ECHO;

    /*
        The <<EOF>>)rule: pop a pushed file, or terminate the lexer
    */
<<EOF>>                 {
                            if (!popSource())
                                yyterminate();
                        }
```

Since the derived class is able to access the information stored within the lexical scanner itself (it can even access the information *directly*, since the data members of `yyFlexLexer` are `protected`, and thus accessible to derived classes), very much processing can be done by the derived class' member functions. This results in a very clean setup of the lexer specification file, in which hardly any code is required in the *preamble*.

### The derived class: Scanner

The `class Scanner` is derived from the `class yyFlexLexer`, generated by `flex++`. The derived class has access to the data controlled by the lexical scanner. In particular, the derived class has access to the following data members:

- `char *yytext`: contains the text matched by a regular expression
- `int yyleng`: the length of the text in `yytext`
- `int yylineno`: the current line number (only if `%option yylineo` was specified in the lexer specfication file)

Other members are available as well, but they are less often used in our experience. Details can be found in the file `FlexLexer.h`, which is part of the `flex` distribution.

The `class Scanner` has to perform two tasks: It should push file information about the current file to a filestack, and should pop the information pushed last once `EOF` is detected on a file.

Several member functions are needed for the accomplishment of these tasks. As they are auxiliary to the `switchSource()` and `popSource()` functions, they are private members. In practice, these private members are developed once the need for them arises. In the following interface of the `Scanner` class the final header file is given. Note that, apart from the private member functions, several private data members are used as well. These members are initialized in the constructor `Scanner()` and are used in the private member functions. They are discussed below, in the context of the member functions using them.

```
#include <FlexLexer.h>  // provides yyFlexLexer interface
#include <fstream.h>
#include <stdio.h>
#include <string.h>

class Scanner: public yyFlexLexer
{
    public:
        Scanner(istream *yyin);

        int yylex();        // overruling yyFlexLexer's yylex()
    private:
        void switchSource();
        int  popSource();
```

```
        int scanYYText(); // 1: nextSource contains new name
        void performSwitch();
        void checkCircularity();
        void checkDepth();

        yy_buffer_state
            **state;
        char
            **fileName,
            *srcPtr,
            *nextSource;

        int
            stackTop;

        static int const
            sizeof_buffer = 16384,
            stackDepth = 10;
};
```

The `switchSource()` member function should interpret the information given in `yytext`: it is interpreted by `scanYYText()`. If `scanYYText()` can extract a filename from `yytext` a switch to another file can be performed. This switch is performed by `performSwitch()`. If the filename could not be extracted, a message is written to the outputstream. Here is the code of `switchSource()`:

```
#include "scanner.h"

void Scanner::switchSource()
{
    if (scanYYText())
        performSwitch();
}
```

The `performSwitch()` function and the matching function `popSource()` handle a simple file switch. In particular, the `yylineno` variable is not updated when a file switch is performed. If line numbers are to be monitored, the `performSwitch()` and `popSource()` functions should respectively push the current value of `yylineno` on a stack, and thereafter reset `yylineno`, and (at `EOF`) pop `yylineno` from the stack.

The member function `scanYYText()` performs a simple scan of the information in `yytext`. If a name is detected following `#include` < that name is stored in the private data member `nextSource`, and 1 is returned. Otherwise, the information in `yytext` is copied to `yyout`, and 0 is returned. Here is the source for `scanYYText()`:

```
#include "scanner.h"

int Scanner::scanYYText()
{
    delete nextSource;          // build new buffer
```

```
    nextSource = new char[yyleng];

    if
    (
        sscanf(yytext, "#include %[^ \t\n>]", nextSource) != 1
        ||
        !(srcPtr = strchr(nextSource, '<'))
    )
    {
        *yyout << yytext;        // copy #include to yyout
        return (0);              // scan failed
    }
    srcPtr++;
    return (1);
}
```

The function `performSwitch()` performs the actual file-switching. The `yyFlexLexer` class provides a series of member functions that can be used for file switching purposes. The file-switching capability of a `yyFlexLexer` object is founded on the `struct yy_buffer_state`, containing the state of the scan-buffer of the file that is currently scanned by the lexical scanner. This buffer is pushed on a stack when an `#include` is encountered, to be replaced with the buffer of the file that is mentioned in the `#include` directive.

The switching of the file to be scanned is realized in the following steps:

- First, the current depth of the `include`-nesting is inspected. If the `stackDepth` is reached, the stack is full, and the program aborts with an appropriate message. For this the member function `checkDepth()` is called.
- Next, the `fileName` stack is inspected, to avoid circular inclusions. If `nextSource` is encountered in the `fileName` array, the inclusion is refused, and the program terminates with an appropriate message. The member function `checkCircularity()` is called for this task.
- Then, a new `ifstream` object is created, assigned to `nextSource`. If this fails, the program terminates with an appropriate message.
- Finally, a new `yy_buffer_state` is created for the newly opened stream, and the lexical scanner is instructed to switch to that stream using `yyFlexLexer`'s member function `yy_switch_to_buffer`.

The sources for the member functions `performSwitch() , checkDepth()`, and `checkCircularity()` are given next:

```
#include "scanner.h"

void Scanner::performSwitch()
{
    ++stackTop;
    checkDepth();
    checkCircularity();
```

```
    ifstream
        *newStream = new ifstream(srcPtr);

    if (!*newStream)
    {
        cerr << "Can't open " << srcPtr << endl;
        exit(1);
    }
    state[stackTop] = yy_current_buffer;
    yy_switch_to_buffer(yy_create_buffer(newStream, sizeof_buffer));
}
```

```
#include "scanner.h"

void Scanner::checkDepth()
{
    if (stackTop == stackDepth)
    {
        cerr << "Inclusion level exceeded. Maximum is " << stackDepth << endl;
        exit (1);
    }
}
```

```
#include "scanner.h"

void Scanner::checkCircularity()
{
    delete fileName[stackTop];

    fileName[stackTop] = new char [strlen(srcPtr) + 1];
    strcpy(fileName[stackTop], srcPtr);

    int
        index;

    for (index = 0; strcmp(srcPtr, fileName[index]); index++)
        ;

    if (index != stackTop)
    {
        cerr << "Circular inclusion of " << srcPtr << endl;
        while (stackTop > index)
        {
            cerr << fileName[stackTop] << " was included in " <<
                    fileName[stackTop - 1] << endl;
            --stackTop;
        }
        exit (1);
    }
}
```

The member function `popSource()` is called to pop the previously pushed sourcefile from the stack, to continue its scan just beyond the just processed `#include` directive. The `popSource()` function first inspects `stackTop`: if the variable is at least 0, then it's an index into the `yy_buffer_state` array, and thus the current buffer is deleted, to be replaced by the state waiting on top of the stack. This is realized by the `yyFlexLexer` members `yy_delete_buffer` and `yy_switch_to_buffer`.

If a previous buffer waited on top of the stack, then 1 is returned, indicating a successful switch to the previously pushed file. If the stack was empty, 0 is returned, and the lexer will terminate.

Here is the source of the function `popSource()`:

```
#include "scanner.h"

int Scanner::popSource()
{
    if (stackTop >= 0)
    {
        yy_delete_buffer(yy_current_buffer);
        yy_switch_to_buffer(state[stackTop]);

        stackTop--;
        return (1);
    }
    return (0);
}
```

These functions complete the implementation of the complete lexical scanner. the lexical scanner itself is stored in the `Scanner::yylex()` function. The `Scanner` object itself only has three public member functions: one function to push a sourcefile on a stack when a switch to the next sourcefile is requested, one function to restore the previously pushed source, and of course `yylex()` itself.

Finally, the constructor will initialize the `Scanner` object. Note that the interface contains an overloaded assignment operator and a copy constructor. By mentioning these two functions in the interface only, without implementing them, they cannot be used in a program: the linking phase of a program using such functions would fail. In this case this is intended behavior: the `Scanner` object does its own job, and there simply is no need for the assignment of a `Scanner` object to another one, or for the duplication of a `Scanner` object.

The constructor itself is a simple piece of code. Here is its source:

```
#include "scanner.h"

Scanner::Scanner(istream *yyin)
{
    switch_streams(yyin, yyout);

    state = new yy_buffer_state * [stackDepth];
    memset(state, 0, stackDepth * sizeof(yy_buffer_state *));

    fileName = new char * [stackDepth];
    memset(fileName, 0, stackDepth * sizeof(char *));
```

```
    nextSource = 0;

    stackTop = -1;
}
```

### The main() function

The main program is a very simple one. As the program expects a filename to start the scanning process at, initially the number of arguments is checked. If at least one argument was given, then a `ifstream` object is created. If this object can be created, then a `Scanner` object is created, receiving the address of the `ifstream` object as its argument. Then the `yylex()` member function of the `Scanner` object is called. This function is inherited from the `Scanner`'s base class `yyFlexLexer`.

Here is the source-text of the main function:

```
/*                              lexer.cc

   A C++ main()-frame generated by C++ for lexer.cc

*/

#include "lexer.h"           /* program header file */


int main(int argc, char **argv)
{
    if (argc == 1)
    {
        cerr << "Filename argument required\n";
        exit (1);
    }

    ifstream
        yyin(argv[1]);

    if (!yyin)
    {
        cerr << "Can't read " << argv[1] << endl;
        exit(1);
    }

    Scanner
        scanner(&yyin);

    scanner.yylex();
    return (0);
}
```

#### Building the scanner-program

The final program is constructed in two steps. These steps are given for a *unix* system, on which `flex++` and the `Gnu` C++ compiler `g++` have been installed:

- First, the lexical scanner's source is created using `flex++`. For this the command

  ```
  flex++ lexer
  ```

  can be given.

- Next, all sources are compiled and linked, using the `libfl.a` library. The appropriate command here is

  ```
  g++ -o scanner *.cc -lfl
  ```

For the purpose of debugging a lexical scanner the rules that are matched and the tokens that are returned are useful information. When `flex++` is called with the `-d` flag, debugging code will be part of the generated scanner. Apart from that, the debugging code must be activated. Assuming the scanner object is called `scanner`, the statement

```
scanner.set_debug(1);
```

must be given following the creation of the `scanner` object.

### 18.4.2 Using both bison++ and flex++

When the input language exceeds a certain level of complexity, a *parser* is generally needed to control the complexity of the input language. In these cases, a *parser generator* is used to generate the code that's required to determine the grammatical correctness of the input language. The function of the scanner is to provided chunks of the input, called *tokens*, for the parser to work with.

Starting point for a program using both a parser and a scanner is the grammar: the grammar is specified first. This results in a set of tokens which can be returned by the lexical scanner (commonly called the *lexer*. Finally, auxiliary code is provided to fill in the blanks: the actions which are performed by the parser and the lexer are not normally specified with the grammatical rules or lexical regular expressions, but are executed by functions, which are called from within the parser's rules or associated with the lexer's regular expressions.

In the previous section we've seen an example of a C++ class generated by `flex++`. In the current section the parser is our main concern. The parser can be generated from a grammar specified for the program `bison++`. The specification of `bison++` is similar to the specifications required for `bison`, but a class is generated, rather than a single function. In the next sections we'll develop a program converting *infix* expressions, in which binary operators are written between their operands, to *postfix* expressions, in which binary operators are written following their operands. A comparable situation holds true for the unary operators `-` and `+`: We can ignore the `+` operator, but the `-` is converted to a unary minus.

Our calculator will recognize a minimal set of operators: multiplication, addition, parentheses, and the unary minus. We'll distinguish real numbers from integers, to illustrate a subtlety in the bison-like grammar specifications, but that's about it: the purpose of this section, after all, is to illustrate a **C++** program, using a parser and a lexer, and not to construct a full-fledged calculator.

In the next few sections we'll start developing the grammar in a `bison++` specification file. Then, the regular expressions for the scanner are specified according to the requirements of `flex++`. Finally the program is constructed.

The class-generating bison software (bison++) is not widely available. The version used by us is 2.20. It can be obtained from

ftp.icce.rug.nl:/pub/unix/bison++2.20.tar.gz.

### The bison++ specification file

The bison specification file used with `bison++` is comparable to the specification file used with `bison`. Differences are related to the class nature of the resulting parser. The calculator will distinguish real numbers from ints, and will support the basic set of arithmetic operators.

The `bison++` specification file consists of the following sections:

- The *header* section. This section is comparable to the **C** specification section used with `bison`. The difference being the `%header`{ opening. In this section we'll encounter mainly declarations: header files are included, and the `yyFlexLexer` object is declared.
- The token section. In this section the bison tokens, and the priority rules for the operators are declared. However, `bison++` has several extra items that can be declared here. They are important and warrant a section of their own.
- The rules. The grammatical rules define the grammar. This section has not changed since the `bison` program.

### The bison++ token section

The token section contains all the tokens that are used in the grammar, as well as the priority rules as used for the mathematical operators. Moreover, several new items can be declared here:

- `%name ParserName`. The name `ParserName` will be the name of the parser's class. This entry should be the first entry of the token-section. It is used in cases where multiple grammars are used, to make sure that the different parser-classes use unique identifiers. By default the name `parse` is used.
- `%define name content`. The `%define` has the same function as the `#define` statement for the `C++` preprocessor. It can be used to define, e.g., a macro. Internally, the defined symbol will be the concatenation of `YY_`, the parser's classname, and the name of the macro. E.g.,

  `YY_ParserName_name`

  Several symbols will normally be defined here. Of all the definitions that can be given here, two are *required*:

– `%define LEX_BODY inline-code`: here the body of the call to the lexer is defined. It can be defined as `= 0` for an abstract parser-class, but otherwise it must contain the code (including surrounding curly braces) representing the call to the lexer. For example, if the lexer object generated by `flex++` is called `lexer`, this declaration should be

```
%define LEX_BODY {return lexer.yylex();}
```

– `%define ERROR_BODY inline-code`: similarly, the body of the code of the call to the error-function is defined here. It can be defined as `= 0`, in which case the parser's class will again become abstract. Otherwise, it is used to specify the inner workings of the error function, including surrounding braces. E.g.,

```
%define ERROR_BODY { cerr << "syntax Error\n"; }
```

When the `LEX_BODY` and `ERROR_BODY` definitions are omitted, then the compiler is not able to complete the virtual table of the parser class, and the linking phase will report an error like

```
undefined reference to 'Parser virtual table'
```

The remaining symbols are optional, and can be (re)defined as needed:

– `%define DEBUG 1`: if non-0 debugging code will be included in the parser's source.

– `%define ERROR_VERBOSE`: if defined, the parser's stack will be dumped when an error occurs.

– `%define LVAL yylval`: the default variable name is shown here: the variable name containing the parser's semantic value is by default `yylval`, but its name may be redefined here.

– `%define INHERIT :public ClassA, public ClassB`: the inheritance list for the parser's class. Note that it starts with the `':'` character. The `define` should be left out if the parser's class isn't derived from another class.

– `%define MEMBERS member-prototypes`: if the parser should contain extra members, they must be declared here. Note that there is only one `%define MEMBERS` definition allowed. So, if multiple members are to be declared, they must all be declared at this point. To prevent very long lines in the specification file, the \ can be used at the end of a line, to indicate that it continues on the next line of the source-text. E.g.,

```
%define MEMBERS void lookup(); void lookdown();
```

  The `MEMBERS` section starts in a `public` section. If private members are required too, a `private:` directive can be part of the `MEMBERS` section.

– Constructor-related defines: When a special parser constructor is needed, then three `%defines` can be used:

  * `%define CONSTRUCTOR_PARAM parameterlist`: this defines the parameterlist for the parser's constructor. Here the types and names of the parameters of the parser should be given. The surrounding parentheses of the parameterlist are not part of the `CONSTRUCTOR_PARAM` definition.
  * `%define CONSTRUCTOR_INIT :initializer(s)`: this defines the base-class and member initializers for the constructor. Note the initial colon following `CONSTRUCTOR_INIT`, which is required. The colon may be given immediately after the `CONSTRUCOR_INIT` statement, or blanks may be used to separate the symbol from the colon.
  * `%define CONSTRUCTOR_CODE { code }`: this defines the code of the parser's constructor.

When the parser doesn't need special effects, a constructor will not be needed. In those cases the parser can be created as follows (using the default parser-name):

```
parse parser;
```

- `%union`. This starts the definition of the semantical value union. It replaces the `#define YYSTYPE` definition seen with `bison`. An example of a `%union` declaration is

```
%union
{
    int
        i;
    double
        d;
};
```

  The union cannot contain objects as its fields, as constructors cannot be called when a union is created. This means that a `string` cannot be a member of the union. A `string *`, however, *is* a possible union member. As a side line: the lexical scanner has no need to know about this union. The scanner can simply pass its scanned text to the parser through its `YYText()` member function. At the appropriate action block a statements like

```
$$.i = atoi(scanner.YYText());
```

  can be used to convert the matched text to a value of an appropriate type.

- Associating tokens and nonterminals with unionfields. Tokens and nonterminals can be associated with unionfields. This is strongly advised. By doing so, the parser's actions-code becomes much cleaner than if the tokens aren't associated with fields. As nonterminals can also be associated with unionfields, the generic returnvariable `$$` or the generic returnvalues `$1, $2`, etc, that are associated with components of rules can be used, rather than `$$.i, $3.d`, etc. To associate a nonterminal or a token with a unionfield, the <`fieldname`> specification is used. E.g.,

```
%token <i> INT          // token association (deprecated)
       <d> DOUBLE
%type  <i> intExpr      // non-terminal association
```

  In this example, note that both the tokens and the nonterminals can be associated with a field of the union. However, as noted earlier, the lexical scanner has no need to know about all this. In our opinion, it is cleaner to let the scanner do just one thing: scan texts. The parser knows what it's all about, and may convert strings like `"123"` to an integer value. Consequently, we are discouraging the association of a unionfield and a token. In the upcoming description of the rules of the grammar this will be further illustrated.

- In the `%union` discussion the `%token` and `%type` specifications should be noted. They are used for the specficiation of the tokens (terminal symbols) that can be returned by the lexical scanner, and for the specification of the returntypes of nonterminals. Apart from `%token` the token-indicators `%left, %right` and `%nonassoc` may be used to specify the associativity of operators. The token(s) 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 precedence. To overrule a certain precedence in a certain context, `%prec` can be used. As all this is standard `bison` practice, it isn't further discussed in this context. The documentation provided with the `bison` distribution should be consulted for further reference.

### The bison++ grammar rules

The rules and actions of the grammar are specified as usual. The grammar for our little calculator is given below. A lot of rules, but they illustrate the use of nonterminals associated with value-types.

```
lines:
    lines
    line
|
    line
;

line:
    intExpr
    '\n'
    {
        cerr << "int: " << $1 << endl;
    }
|
    doubleExpr
    '\n'
    {
        cerr << "double: " << $1 << endl;
    }
|
    '\n'
    {
        cout << "Good bye\n";
        YYACCEPT;
    }
|
    error
    '\n'
;

intExpr:
    intExpr '*' intExpr
    {
        $$ = $1 * $3;
    }
|
    intExpr '+' intExpr
    {
        $$ = $1 + $3;
    }
```

```
|
    '(' intExpr ')'
    {
        $$ = $2;
    }
|
    '-' intExpr         %prec UnaryMinus
    {
        $$ = -$2;
    }
|
    INT
    {
        $$ = atoi(lexer.YYText());
    }
;

doubleExpr:
    doubleExpr '*' doubleExpr
    {
        $$ = $1 * $3;
    }
|
    doubleExpr '+' doubleExpr
    {
        $$ = $1 + $3;
    }
|
    doubleExpr '*' intExpr
    {
        $$ = $1 * $3;
    }
|
    doubleExpr '+' intExpr
    {
        $$ = $1 + $3;
    }
|
    intExpr '*' doubleExpr
    {
        $$ = $1 * $3;
    }
|
    intExpr '+' doubleExpr
    {
        $$ = $1 + $3;
    }
|
    '(' doubleExpr ')'
    {
        $$ = $2;
    }
|
```

```
    '-' doubleExpr          %prec UnaryMinus
    {
        $$ = -$2;
    }
|
    DOUBLE
    {
        $$ = atof(lexer.YYText());
    }
;
```

With these rules a very simple calculator is defined in which integer and real values can be negated, added, and multiplied, and in which standard priority rules can be circumvented using parentheses. The rules show the use of typed nonterminal symbols: `doubleExpr` is linked to real (double) values, `intExpr` is linked to integer values. Precedence and type association is defined in the token section of the parser specification file, which is:

```
%name  Parser
%union
{
    int i;
    double d;
};
%token      INT
            DOUBLE
%type   <i> intExpr
        <d> doubleExpr

%left   '+'
%left   '*'
%right  UnaryMinus

%define MEMBERS                                 \
    virtual ~Parser()   {}                      \
    private:                                    \
        yyFlexLexer lexer;
%define LEX_BODY {return lexer.yylex();}

%define ERROR_BODY { cerr << "error encountered\n"; }
```

In the token section we see the use of the `%type` specifiers, connecting `intExpr` to the `i`-field of the semantic-value union, and connecting `doubleExpr` to the `d`-field. At first sight it looks a bit complex, since the expression rules must be included for each individual returntype. On the other hand, if the union itself would have been used, we would have had to specify somewhere in the returned semantic values what field to use: less rules, but more complex and error-prone code.

Also, note that the lexical scanner is included as a member of the parser. There is no need to define the scanner outside of the parser, as it's not used outside of the parser object. The virtual destructor is included as an member to prevent the compiler from complaining about the parser having a non-virtual destructor.

### The flex++ specification file

The flex-specification file to be used with our little calculator is simple: blanks are skipped, single characters are returned, and numerical values are returned as either `Parser::INT` or `Parser::DOUBLE` values. Here is the complete `flex++` specification file:

```
%{
#include "parser.h"
%}

%%

[ \t]                       ;
[0-9]+                      return(Parser::INT);

"."[0-9]*                   |
[0-9]+("."[0-9]*)?          return(Parser::DOUBLE);

exit                        |
quit                        return (Parser::DONE);

.|\n                        return (*yytext);
```

### The generation of the code

The code is generated in the same way as with `bison` and `flex`. To order `bison++` to generate the files `parser.cc` and `parser.h`, the command

```
bison++ -d -o parser.cc parser
```

can be given.

`Flex++` will thereupon generate code on `lexer.cc` using the command

```
flex++ -I -olexer.cc lexer
```

Note here that `flex++` expects no blanks between the `-o` flag and `lexer.cc`.

On *unix*, linking and compiling the generated sources and the source for the main program (listed below) is realized with the following command:

```
g++ -o calc -Wall *.cc -lfl -s
```

Note the fact that the `libfl.a` library is mentioned here. If it's not mentioned unresolved functions like `yywrap()` emerge.

A source in which the `main()` function, the lexical scanner and the parser objects are defined is, finally:

```
#include "parser.h"
int main()
{
    Parser
        parser;
    return (parser.yyparse());
}
```