C++ Annotations Version 4.2.4

Frank B. Brokken and Karel Kubat
ICCE, University of Groningen
Westerhaven 16, 9718 AW Groningen
Netherlands
Published at the University of Groningen
ISBN 90 367 0470 7

1994 - 1997

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/C++** programming courses, which are given yearly at the University of Groningen. As such, this document is not a complete **C/C++** handbook. Rather, it serves as an addition to other documentation sources.

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

Contents

1 Overview of the chapters **8**

2 Introduction **10**

2.0.1 History of the C++ Annotations 11

2.1 The history of C++ 12

2.1.1 Compiling a C program by a C++ compiler 12

2.2 Advantages and pretensions of C++ 14

2.3 What is Object-Oriented Programming? 15

2.4 Differences between C and C++ 17

2.4.1 End-of-line comment 17

2.4.2 NULL-pointers vs. 0-pointers 18

2.4.3 Strict type checking 18

2.4.4 The void argument list 19

2.4.5 The #define __cplusplus 19

2.4.6 The usage of standard C functions 20

2.4.7 Header files for both C and C++ 21

2.4.8 The definition of local variables 22

2.4.9 Function Overloading 23

2.4.10 Default function arguments 25
2.4.11 The keyword typedef 27
2.4.12 Functions as part of a struct 28

3 A first impression of C++ **30**

3.1 More extensions of C in C++ 30
3.1.1 The scope operator :: 30
3.1.2 cout, cin and cerr 31
3.1.3 The 'bool' data type 33
3.1.4 The 'wchar_t' data type 35
3.1.5 The keyword const 35
3.1.6 References 37
3.2 Functions as part of structs 43
3.3 Data hiding: public, private and class 44
3.4 Structs in C vs. structs in C++ 47

4 Classes **50**

4.1 Constructors and destructors 51
4.1.1 The constructor 51
4.1.2 The destructor 54
4.1.3 A first application 55
4.1.4 Constructors with arguments 60
4.2 Const member functions and const objects 63
4.3 The operators new and delete 65
4.3.1 Allocating and deallocating arrays 65
4.3.2 New and delete and object pointers 69
4.3.3 The function set_new_handler() 71

4.4 The keyword inline 73
4.4.1 Inline functions within class declarations 74
4.4.2 Inline functions outside of class declarations 75
4.4.3 When to use inline functions 75
4.5 Objects in objects: composition 76
4.5.1 Composition and const objects: const member initializers 78
4.5.2 Composition and reference objects: reference member initializers 80
4.6 Friend functions and friend classes 82
4.7 Header file organization with classes 84

5 Classes and memory allocation 91

5.1 Classes with pointer data members 92
5.2 The assignment operator 94
5.2.1 Overloading the assignment operator 97
5.3 The this pointer 101
5.3.1 Preventing self-destruction with this 102
5.3.2 Associativity of operators and this 104
5.4 The copy constructor: Initialization vs. Assignment 105
5.4.1 Similarities between the copy constructor and operator=() 111
5.5 Conclusion 114

6 More About Operator Overloading 115

6.1 Overloading operator[]() 115
6.2 Overloading operator new(size_t) 119
6.3 Overloading operator delete(void *) 123
6.4 Cin, cout, cerr and their operators 124

6.5 Conversion operators 125
6.6 Overloadable Operators 129

7 Static data and functions **131**

7.1 Static data 132
7.1.1 Private static data 132
7.1.2 Public static data 135
7.2 Static member functions 136

8 Classes having pointers to members **138**

8.1 Pointers to members: an example 138
8.2 Initializing pointers to members 139
8.3 Pointers to static members 141
8.4 Using pointers to members for real 142
8.4.1 Pointers to members: an implementation 147

9 The IO-stream Library **154**

9.1 Iostreams: insertion (<<) and extraction (>>) 155
9.1.1 The insertion operator << 155
9.1.2 The extraction operator >> 157
9.2 Four standard iostreams 159
9.3 Files in general 159
9.3.1 Writing streams 160
9.3.2 Reading streams 161
9.3.3 Reading and writing streams 163
9.3.4 IOStream Condition States 165
9.3.5 Special functions 166

9.3.6 Formatting 168

10 More about friends **175**

10.1 Inserting String objects into streams 177

10.2 An initial solution 178

10.3 Friend-functions 180

10.3.1 Preventing the friend-keyword 181

10.4 Friend classes 182

11 Inheritance **185**

11.1 Related types 186

11.2 The constructor of a derived class 190

11.3 Redefining member functions 191

11.4 Multiple inheritance 194

11.5 Conversions between base classes and derived classes 197

11.5.1 Conversions in object assignments 197

11.5.2 Conversions in pointer assignments 199

11.6 Storing base class pointers 200

12 Polymorphism, late binding and virtual functions **204**

12.1 Virtual functions 205

12.1.1 Polymorphism in program development 208

12.1.2 How polymorphism is implemented 210

12.2 Pure virtual functions 211

12.3 Comparing only Persons 214

12.4 Virtual destructors 216

12.5 Virtual functions in multiple inheritance 217

12.5.1 Ambiguity in multiple inheritance 218
12.5.2 Virtual base classes 220
12.5.3 When virtual derivation is not appropriate 222

13Exceptions **226**

13.1 Using exceptions: an outline 227
13.1.1 Compiling sources in which exceptions are used 228
13.2 An example using exceptions 228
13.2.1 No exceptions: the setjmp() and longjmp() approach . . . 231
13.2.2 Exceptions: the preferred alternative 234
13.3 Throwing exceptions 236
13.3.1 The empty throw statement 241
13.4 The try block 243
13.5 Catching exceptions 244
13.5.1 The default catcher 246
13.6 Declaring exception throwers 248

14Templates **250**

14.1 Template functions 250
14.2 Template classes 252
14.2.1 A template class: Array 252
14.2.2 Using the Array class 256
14.3 Templates and Exceptions 257
14.4 Evaluation of template classes 258

15Concrete examples of C++ **260**

15.1 Storing objects: Storable and Storage 260

15.1.1 The global setup 261
15.1.2 The class Storable 264
15.1.3 The class Storage 266
15.2 A binary tree 270
15.2.1 The Node class 270
15.2.2 The Tree class 272
15.2.3 Using Tree and Node 277
15.3 Classes to process program options 281
15.3.1 Functionality of the class Configuration 281
15.3.2 Implementation of the class Configuration 285
15.3.3 The class Option 290
15.3.4 Derived from Option: The class TextOption 294
15.3.5 The class Object 296
15.3.6 The class Hashtable 297
15.3.7 Auxiliary classes 302
15.4 Using Bison and Flex 309
15.4.1 Using Flex++ to create a scanner 309
15.4.2 Using both bison++ and flex++ 320

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 'class' concept: structs having functions. The 'object' concept: variables of a `class`.
- Chapter 5: Allocation and returning unused memory: `new`, `delete`, and the function `set_new_handler()`.
- Chapter 6: More About Operator Overloading.
- Chapter 7: Static data and functions: components of a class not bound to objects.
- Chapter 8: Classes having pointer members: how to prevent memory leaks and wild pointers.
- Chapter 9: The **C++** type-safe I/O library.
- Chapter 10: Gaining access to private parts from outside: friend functions and classes.
- Chapter 11: Building classes upon classes: setting up class hierarcies.
- Chapter 12: Polymorphism: changing the behavior of memberfunctions accessed through base class pointers.

- Chapter 13: Exceptions: handling errors where appropriate, rather than where they occur.
- Chapter 14: Templates: using *molds* for code that is type dependent.
- Chapter 15: 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 take note of the fact 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 4.2.4) 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., typos are corrected.

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 3.1.3 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

[1] e.g., the Dutch book *De programmeertaal C*, Brokken and Kubat, Univerity of Groningen 1996, or the Microsoft **C/C++** tutorial

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++**.

This document is published by the ICCE, University of Groningen, the Netherlands. This document was typeset using the yodl formatting system.

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

Frank B. Brokken ICCE University of Groningen PO Box 335, 9700 AH Groningen The Netherlands (email: frank@icce.rug.nl)

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

2.0.1 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: `frank@icce.rug.nl`.

2.1 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.

2.1.1 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 3.1.4), while an `int` is still an `int`. As we shall see later (see section 2.4.9), 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 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 11).

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!

2.3 What is Object-Oriented Programming?

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

In contrast, or maybe better: in addition to this, an object-oriented approach identifies the **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

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

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.`

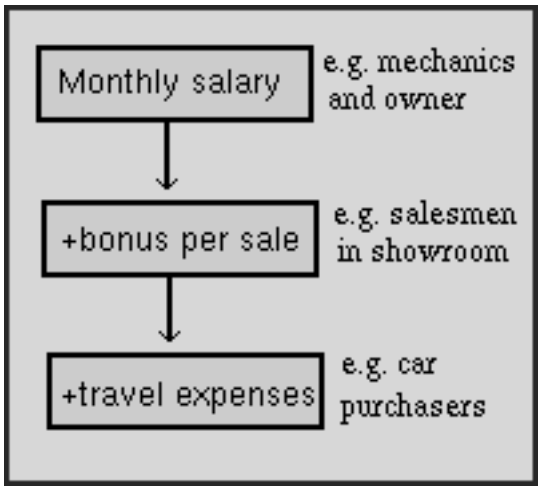
When representing the above salary administration, the keywords could be mechanics, owner, salesmen and purchasers. The properties of such units are: a monthly salary, sometimes a bonus per purchase or sale, and sometimes restitution of travel expenses. When analyzing the problem in this manner we arrive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



\

Figure 2.1: Hierarchy of objects in the salary administration.

types, more complex types can be derived *ad infinitum*, until a representation of the entire problem can be made.

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

2.4 Differences between C and C++

In this section some examples of **C++** code are shown. Some differences between **C** and **C++** are highlighted.

2.4.1 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.

2.4.2 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.

2.4.3 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.

2.4.4 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.

2.4.5 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.

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

2.4.6 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.

2.4.7 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.

There is more to be said about header files. In section 4.7 the preferred organization of header files when **C++** classes are used is discussed.

2.4.8 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.

2.4.9 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.

2.4.10 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:
```

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

```
                                // 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:

```
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.

2.4.11 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;
```

2.4.12 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 2.3), 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 `structs` 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 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.h>

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.h`.
- 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 not part of **C++** grammar *sec*, as defined in the compiler which parses source files. The streams are part of the definitions

in the header file `iostream.h`. 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 9 *iostreams* will be covered in greater detail.

3.1.3 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 a two extra types, the types `bool` and `wchar_t` 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 and `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 begind 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.

3.1.4 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....

3.1.5 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++`.

3.1.6 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()
```

[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.

```
{
    int
        x;

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

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

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

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

int main ()
{
    int
        x;

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

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

  Consider the following example:

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

Person
   person[50];                         // database of persons

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 5

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 2.4.12). 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 Data hiding: public, private and class

As mentioned previously (see section 2.2), 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.

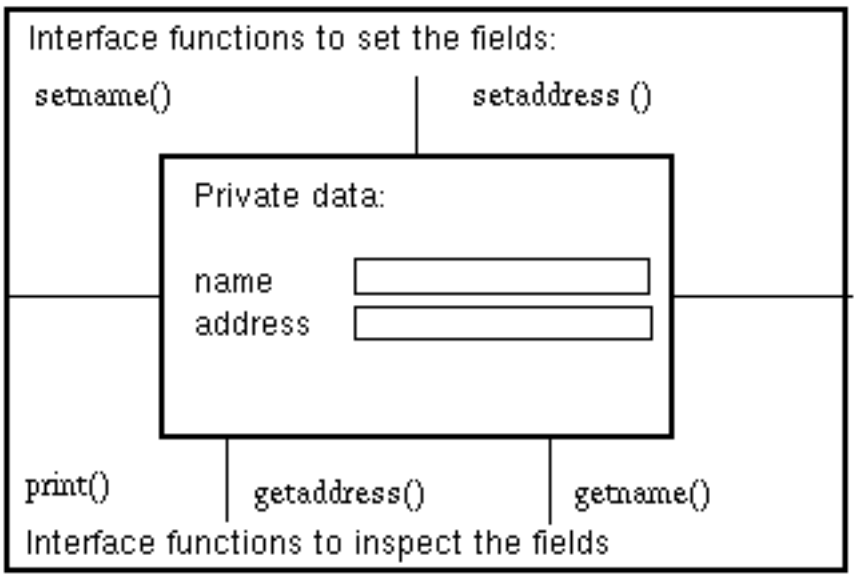
With this knowledge we can expand the `struct person`:

```
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
```

[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.

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:

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

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

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

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

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

3.4 Structs in C vs. structs in C++

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

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

// some functions to manipulate PERSON_ structs

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

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

// etc..
```

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

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

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

```
PERSON_
    x;

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

becomes in **C++**:

```
Person
    x;

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

Chapter 4

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.

4.1 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.

4.1.1 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].

[1] A compiler-supplied constructor in a class which contains composed objects (see section 4.5) will 'automatically' call the member initializers, and therefore does perform some actions.

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 when the function is called.

When an object is a global variable or a static 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.h>

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

Test::Test()                // here is the
{                           // definition
    puts("constructor of class Test called");
}

// 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()
```

We postpone the discussion of such constructors to 4.5.1.

```
    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; it is therefore also called the *default constructor*. This is however no requirement *per se*. We shall later see that it is possible to define constructors with arguments.

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()
```

4.1.2 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 itseld 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.

4.1.3 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].

```
// 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. 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.

[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 `STRDUPNEW`, `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.

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 4.2, 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);
}
```

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.h>

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 ** "`.

4.1.4 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
```

4.2 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 3.1.5 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.

4.3 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 usage of these operators is given below. A pointer variable to an `int` is used to point to memory to which is allocated by `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, such as is the case with 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).

4.3.1 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.h>

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.h>

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.h>

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 4.5.

4.3.2 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.

4.3.3 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[3].
- 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

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

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[4]:

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

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

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

    // install error function
    set_new_handler(out_of_memory);

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

    return (0);
}
```

[4]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*

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 `STRDUPNEW`. It is strongly suggested to revert to this approach, rather than to using functions like `xstrdup()`, when the allocation of memory is required.

4.4 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.

4.4.1 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, and shows that `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.

4.4.2 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.

4.4.3 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()`).
- 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.

4.5 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 `structs` or `unions` 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.

4.5.1 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.

4.5.2 Composition and reference objects: reference member initializers

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

A program uses an object of the class `Configfile`, defined in `main()` to access the information in a configuration file. The configuration file contains parameters of the program which may be set by changing the values in the configuarion 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 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`. Actually, the whole purpose of the statement `conf_ref = conf` was after all 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.

4.6 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 10 we'll continue the discussion, having covered, by that time, the topic of operator overloading.

4.7 Header file organization with classes

In section 2.4.7 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 memberfunctions of classes. With source files there are basically two approaches:

- All required header files for a memberfunction are included in each individual source file.

- All required header files for all memberfunctions 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 `String::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 now 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 11) the hader 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 `ifnef` and `endif` directives.
- Header files of classes of objects that are either composed or inherited (see chapter 11) 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.h>    // 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
```

Chapter 5

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 4.3.3) 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.

5.1 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.

5.2 The assignment operator

Variables which are `struct`s or `class`es can be directly assigned in C++ in the same way that `struct`s 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 5.1.

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.

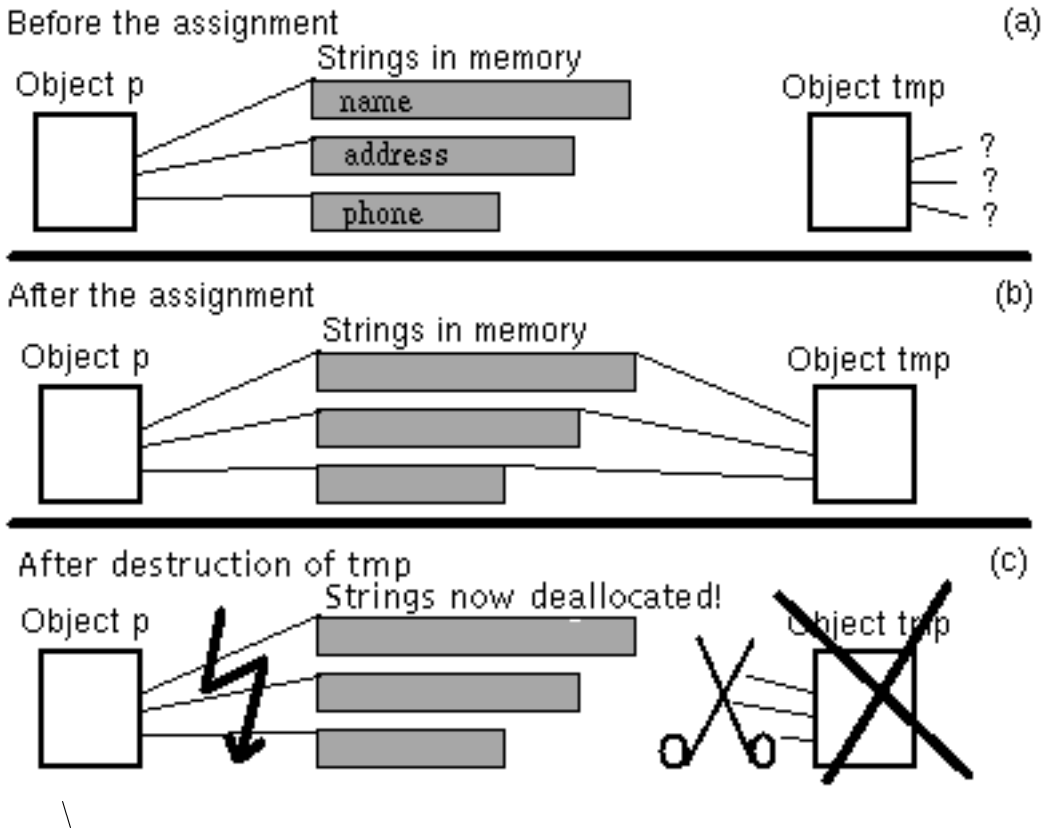
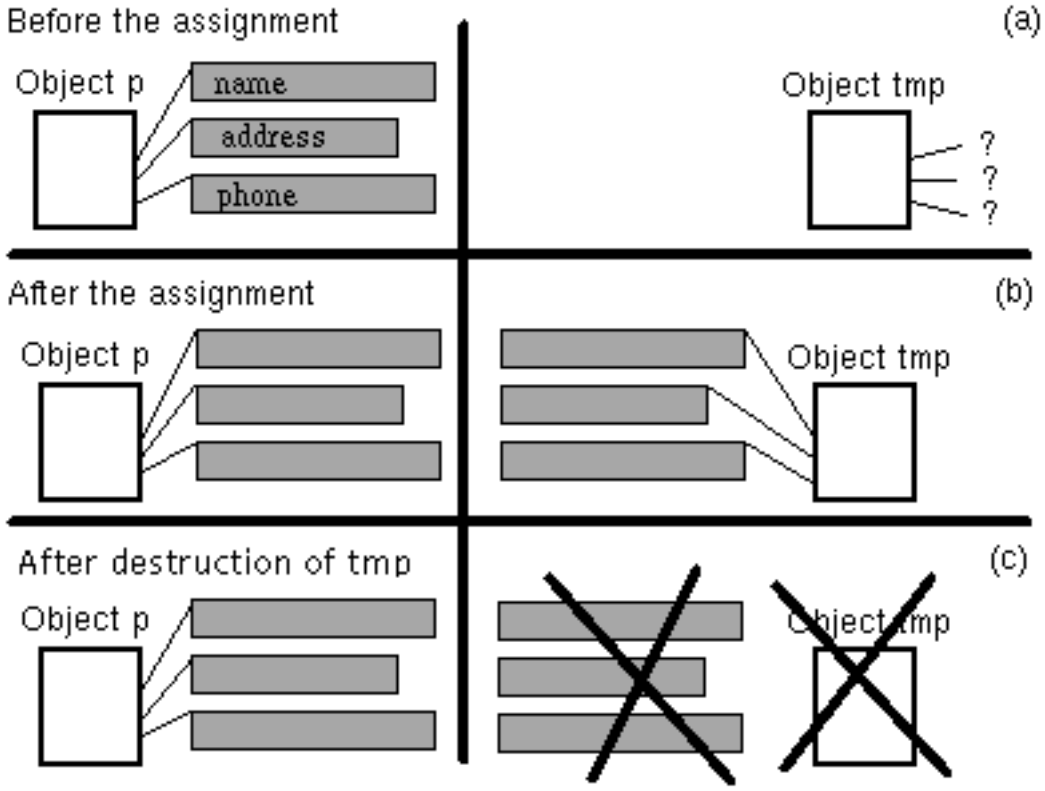


Figure 5.1: Private data and public interface functions of the class Person, using bytewise assignment



\

Figure 5.2: Private data and public interface functions of the class Person, using the 'correct' assignment.

5.2.1 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.

The 'right' way to duplicate a `Person` object is illustrated in figure 5.2.

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 overload-*

ing: 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.

5.3 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:

```
// 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;
```

[1]Note that '`this`' is not available in the not yet discussed `static` member functions.

```
    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.

5.3.1 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`.

5.3.2 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);
}
```

5.4 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 that 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.

5.4.1 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()`.

5.5 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, an overloaded assignment function and a copy constructor should be implemented.

Chapter 6

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.

6.1 Overloading operator[]()

As our next example of operator overloading, we present a class which is meant to operate on an array of `int`s. Indexing the array elements occurs with the standard array operator `[]`, but additionally the class checks for boundary overflow. Furthermore, the 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 `int`s. 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 destroy();             // standard functions
                                    // used to copy/destroy
        void copy(IntArray const &other);

        int
            *data,
            size;
};
```

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* a `int const &`, but rather a `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 5.4.1), 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.

```
IntArray::IntArray(int sz)
{
    if (sz < 1)
    {
        cout << "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
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 overloaded array operator
int &IntArray::operator[](int index)
{
    // check for array boundary over/underflow
    if (index < 0 || index >= size)
    {
        cout << "IntArray: boundary overflow or underflow, index = "
             << index << ", should range from 0 to " << size - 1 << endl;
        exit(1);
    }
    return (data[index]);   // emit the reference
}
```

6.2 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, what happens when the operator `new` is defined for the `class X`, assuming that class is defined as follows[1]:

```
class X
{
    public:
        void *operator new(size_t sizeofX);

        int
            x,
            y,
            z;
};
```

Next, 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;
    return (0);
}
```

[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`.

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 memberfunction. 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 (discouraged 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 *`.

6.3 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;
}
```

6.4 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 <<. Adaptation of a class for the usage with `cin` and its extraction operator >> occurs in a similar way and is not 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.h` 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 5.1) , 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 9).

  An improved form of the above function would therefore be:

```
ostream &operator<<(ostream &stream, Person const &pers)
{
    if (! stream.opfx())
        return (stream);
    ...
}
```

6.5 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
```

[2]Note that the assingment from a `char const *` also includes the null-pointer. An assignment like `stringObject = 0` is perfectly in order.

```
            n;
};
```

The `StringArray` class has one interesting memberfunction: 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 6.6). 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 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", (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;
};
```

6.6 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 6.5.

Chapter 7

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.

7.1 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.

7.1.1 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 memberfunction 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.

7.1.2 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 7.1 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];
```

7.2 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: in that case, the function can only be called from other member functions but not from 'outside of the class'.

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 8

Classes having pointers to members

Pointers in classes have been discussed in detail in chapter 5.1. 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.

8.1 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 memberfunction 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 memberfunction `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.

8.2 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 memberfunction of the class `String`, while such a memberfunction 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.

8.3 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 memberfunction 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 memberfunction `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 memberfunction

    cout << pfi() << endl;
                // print the value produced by
                // String::n_strings()
}
```

8.4 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:
        ...
        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 organisation of `Person` and `Person_dbase` is pictured in figure 8.1: Within a `Person_dbase` object the `Person` objects are stored. They can be reached via the pointer variable `Person *pp`.

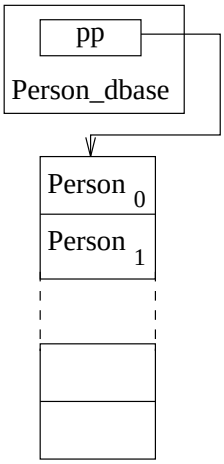
We would like to develop the function `Person_dbase::list()` in such a way that it lists the contents of the database sorted according to a selected field of a `Person` object.

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

[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,



\

Figure 8.1: Person_dbase objects: Persons reached via Person *pp

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:

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`.

```
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.

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-memberfunctions 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 memberfunctions 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 memberfunction 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.

8.4.1 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 `int`s (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 `pa` (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 `int`s, `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,
```

[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.

```
    };

    // ... 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
        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 9

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 should be used.

Finally, in order to use the `iostream` facilities, the header file `iostream.h` must be included in source files using these facilities.

9.1 Iostreams: insertion (<<) and extraction (>>)

The insertion and extraction operators are used to write information to or read information from, respectively, *ostream* and *istream* objects.

9.1.1 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 10.3 and 10.3.1).

Consider the following code example:

```
#include <iostream.h>

int main()
{
    int
        value = 15,
        *p = &value;

    cout << "Value: " << value << "\n"
        << "via p: " << *p << "\n"
        << "value's address: " << &value << "\n"
        << "address via p:   " << p << "\n"
        << "p's address:     " << &p << "\n";
}
```

In this form the following output is generated (gnu C++ compiler, version 2.7.2):

```
Value: 15
via p: 15
value's address: 1
address via p:   1
p's address:     1
```

This is a bit unexpected. How to get the addresses? By using an explicit cast to the *generic pointer* `void *` the problem is solved:

```
#include <iostream.h>

int main()
{
    int
        value = 15,
        *p = &value;

    cout << "Value: " << value << "\n"
        << "via p: " << *p << "\n"
        << "value's address: " << (void *)&value << "\n"
        << "address via p:   " << (void *)p << "\n"
        << "p's address:     " << (void *)&p << "\n";
}
```

The above code produces, e.g.,

```
Value: 15
via p: 15
value's address: 0x804a1e4
address via p:   0x804a1e4
p's address:     0xbffff9fc
```

9.1.2 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 to be given new values, but references (with the exception of the `char *`).

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.

9.2 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 als 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.

9.3 Files in general

In order to be able to create `fstream` objects, two header files must be included: `iostream.h` and `fstream.h`. 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.

Apart from the `iostream.h` header file the headerfile `fstream.h` must be included when an `fstream, ofstream,` or `ifstream` object must be constructed or used.

9.3.1 Writing streams

In order to be able to write to a file an `ofstream` object must be created, the constructor receiving 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.h>
#include <fstream.h>

int main()
{
    ofstream
```

```
        out("/");   // creating 'out' fails

    if (!out)
    {
        cerr << "creating ofstream object failed\n";
        exit(1);
    }
}
```

9.3.2 Reading streams

In order to be able to read to a file an `ifstream` object must be created, the constructor receiving 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);
```

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;
```

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];

cin >> 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.h>
#include <fstream.h>

int main()
{
    ifstream
        in("/");   // creating 'in' fails

    if (!in)
    {
        cerr << "creating ifstream object failed\n";
        exit(1);
    }
}
```

9.3.3 Reading and writing streams

In order to be able to read and write to a file an `fstream` 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.

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()` memberfunctions. The memberfunction `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 9.3.4) 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()` memberfunction. 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
```

9.3.4 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 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 .
- `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.

9.3.5 Special functions

Apart from the functions discussed so far, and the extraction and assignment operators, several other functions are available for `stream` objects which are worthwhile mentioning.

- `gcount()`: this function returns the number of characters read by `getline()` (described below) or `read()` (described below).
- `flush()`: this function flushed the output of the `ostream` object.
- `get()`: returns the next character as an `int`: End-of-file is returned as , a value which can't be a character.
- `get(char c)`: this function reads a `char` from an `istream` object, and returns the `istream` object for which the function was called.
  The `get()` and `get(char c)` functions read separate characters, and will not skip whitespace.

- `getline(char *buffer, int size, int delimiter = '\n')`: this function reads up to `size - 1` characters or until `delimiter` was read into `buffer`, and appends a final `ascii-z`. The delimiter is not entered into `buffer`. The function changes the state of the output-stream to *fail* if a line was not terminated by the delimiter. Since this situation will prevent the function from reading more information, the function `clear` must be called in these circumstances to allow the function to produce more information. The frame for reading lines from an `istream` object is, therefore:

```
#include <iostream.h>

int main()
{
    char
        buffer[100];

    while (1)
    {
        cin.getline(buffer, 100);
        cout << buffer;
        if (cin.eof())
            return(0);

        if (cin.good())
            cout << endl;
        else
            cin.clear();
    }
}
```

- `istream &ignore([int n] [, int delimiter])`. This function skips over a certain number of characters, but not beyond the `delimiter` character. By default, the `delimiter` character is : the function `ignore()` will not skip beyond . If the number of characters isn't specified, one character will be skipped.
- `int peek()`. This function returns the character that will be read with the next call to the function `get()`.
- `istream &putback(char c)`. This function attempts to puts character `c` back into the stream. The most recently read character character may always be returned into the stream. If the character can't be returned, is returned. This function is the analogon of **C**'s `ungetc()` function.

- `int opfx()`. This function should be called before any further processing. If the `ostream` object is in the state 'good', `flush()` is called for that object, and 1 is returned. Otherwise, 0 is returned. The `p` in `opfx()` indicates *prefix*: the function should be called before processing the `ostream` object.

- `int osfx()`: This function is the *suffix* equivalent for `opfx()`. called at the conclusion of any processing. All the `ostream` methods end by calling `osfx()`.
  If the `unitbuf` flag is set for this stream, `osfx()` flushes any buffered output for it, while any output buffered for the `C` output streams `stdout` and `stderr` files is flushed if the `stdio` flag was set for this stream.

- `read(char *buffer, int size)`: this function reads `size` bytes from the `istream` object calling this memberfunction into `buffer`.

- `write(char const *str, int length)`: writes `length` characters in `str` to the `ostream` object for which it was called, and it returns the `ostream` object.

9.3.6 Formatting

While the insertion and extraction operators provide elegant ways to read information from and write information to `iostreams`, there are situations in which special formatting is required. Formatting may involve the control of the width of an output field or an input buffer or the form (e.g., the radix) in which a value is displayed. The functions `(v)form()` and `(v)scan()` can be used for special formatting.

Apart from these memberfunctions, memberfunctions 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 memberfunction `form()` is the analogon of **C**'s `fprintf()` function. When variadic functions are constructed in which information must be inserted into a stream, the memberfunction `vform()` can be used, being the analogon to `vfprintf()`.

To scan information from a stream, the meemberfunction `scan()` can be used, which is the analogon of **C**'s `fscanf()` function. Similarly to `vfscanf()`, the memberfunction `vscan()` can be used in variadic functions.

Format states: dec, hex, oct manipulators

The `iostream` objects maintain *format states* controlling the default formatting of values. The format states can be controlled by memberfunctions and by manipulators. Manipulators are inserted into the stream, the memberfunctions are used by themselves.

The manipulators are `dec`, `hex` and `oct`, enforcing 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
```

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: `precision()` is not a manipulator, but a memberfunction. Therefore, `cout.precision()` rather than `precision()` is inserted into the stream.

Settng the display form: the member 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::showbase)
```

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
```

As a summary:

- `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.

The manipulator setw()

The `setw()` manipulator expects one argument: the width of the field that's 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;
```

The 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.h` must be included.

String Formatting

Strings can be processed similarly to `iostream` objects, if objects of the class `istrstream` or `ostrstream` are constructed. Objects of these classes read information from memory and write information to memory, respecttively. These objects are created by constructors expecting the address of a block of memory (and its size) as its argument. For example 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 << '\0';

cout << os.str();       // display the string
```

Note the final `'\0'` character (the ascii-z) that is appended: `ostrstream` objects do their own bookkeepping, and also accept non ascii-z terminated information. Therefore, an ascii-z character must be appended to the string when it is to be inserted into an ostream.

Note also the use of the memberfunction `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.

The following memberfunctions 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 memberfunction 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.
- `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.h` header-file must be included.

Chapter 10

More about friends

Let's return to `friend`s once more. In section 4.6 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.h>

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.

10.1 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

```
void main()
{
    String
        str("Hello world");

    cout << "The string is: '<< str << "'" << endl;
}
```

Analogously, with the extraction operator, we would like to be able to write code comparable to the next example:

```
void main()
{
    String
        str;
```

```
    cout << "Enter your string: ";

    cin >> str;

    cout << "Got: '<< str << "'" << endl;
}
```

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.

10.2 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 >> buf;      // extraction

    delete str;     // free this->str
                    // memory
```

```
                    // assign new value
    str = strdupnew(buf);

    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-exptressions: 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.

10.3 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 &str);
```

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 prevent.

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.h>

class String
{
    friend istream &operator>>(istream &lvalue,
                               String &rvalue);
    public:
        // ...
    private:
        char
            *str;
```

```
};

istream &operator>>(istream &lvalue, String &rvalue)
{
    char
        buffer[500];

    lvalue >> buf;      // extraction

    delete rvalue.str;  // free old 'str' memory

    rvalue.str = strdupnew(buf);
                // assign new value

    return (lvalue);    // 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.

10.3.1 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

memberfunctions 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 >> buf;      // extraction

    rvalue = buf;       // 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.

10.4 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 somehat 'excentric' 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 11

Inheritance

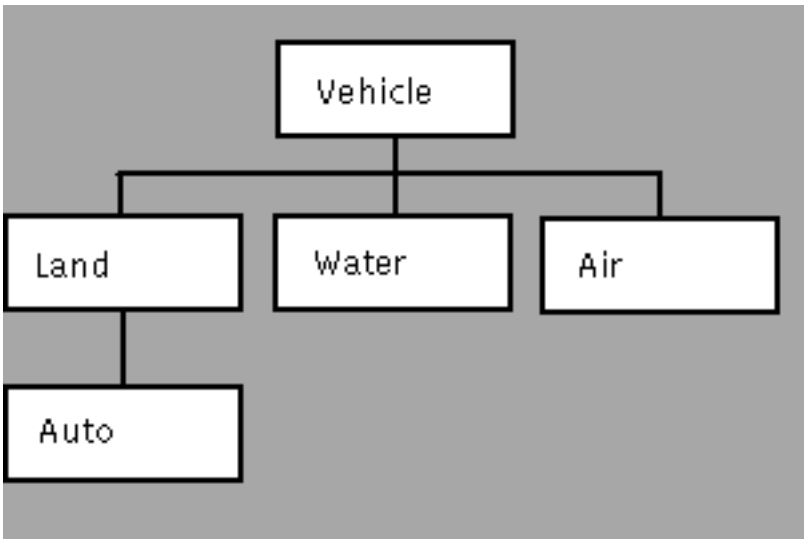
When programming in **C**, it is common to view problem solutions from a *top-down* approach: functions and actions of the program are defined in terms of sub-functions, which again are defined in sub-sub-functions, etc.. This yields a hierarchy of code: `main()` at the top, followed by a level of functions 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 4.5), 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 in-



\

Figure 11.1: Initial object hierarchy of vehicles.

troductory chapter (see section 2.3), 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 11.1.

11.1 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 classi-

fication 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].

[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.

To illustrate the use of the derived class `Land` consider the following example:

```
Land
    veh(1200, 145);

void main()
{
    cout << "Vehicle weighs " << veh.getweight() << endl
         << "Speed is " << veh.getspeed() << endl;
}
```

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 so that the principle of data hiding thus remains ensured. 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 memberfunctions of the parent class `Vehicle` in the class `Land`. We might have to recompile the `Vehicle` sources, though, as the relative locations of the data members within the `Vehicle` objects will have changed due to the modification of the `Land` 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 *memberfunctions* to the base class doesn't alter the data organization, no such recompilation is needed after adding new *memberfunctions*.

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
{
    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 *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.

11.2 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 4.5).

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);
}
```

11.3 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`: for a `Truck` only a `setweight()` function with two `int` arguments can be used.

However, note that the `Vehicle`'s `setweight()` function remains available: beacause of *function overloading* the single argument `setweight()` function still refers to the `Vehicle` part of the `Truck`. This feature is used below in the implementation of the function `Truck::setweight()`.

- 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 following code fragment presents the redefined function `Truck::setweight()`:

```
void Truck::setweight(int engine_wt, int trailer_wt)
{
    trailer_weight = trailer_wt;
    setweight(engine_wt);       // uses Auto::setweight()
}
```

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
        );
}
```

Note that in the function `Truck::setweight()` the single argument function `setweight(engine_wt)` is used without causing ambiguity: only the class `Auto` has such a memberfunction. In the function `Truck::getweight()`, however, the explicit call `Auto::getweight()` is required to select the `getweight()` function of the class `Auto`. An implementation like

```
return (getweight() + trailer_weight);
```

would not be correct: this statement would lead to infinite recursion, and hence to an error in the program execution.

11.4 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.

11.5 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 to such objects are used. Both situations will be discussed next.

11.5.1 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
    .
}
```

11.5.2 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.

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.

11.6 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 `Vehicle`s. The actual pointers may be addresses of `Vehicle`s themselves, but also may refer to derived types such as `Auto`s.

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 &vp);
            // 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
    *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 12

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 11.6 using the storage class `VStorage`, will incorrectly compute the combined weight when a `Truck` object (see section 11.3) 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*.

12.1 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 11.1 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);
```

[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?"

```
    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).

12.1.1 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:

```
typedef enum                /* type of the vehicle */
{
    is_vehicle,
    is_land,
    is_auto,
    is_truck,
} Vtype;

typedef struct              /* generic vehicle type */
{
    int weight;
} Vehicle;

typedef struct              /* land vehicle: adds speed */
{
    Vehicle v;
    int speed;
} Land;

typedef struct              /* auto: Land vehicle + name */
{
    Land l;
    char *name;
} Auto;
```

```
typedef struct              /* truck: Auto + trailer */
{
    Auto a;
    int trailer_wt;
} Truck;

typedef union               /* all sorts of vehicles in 1 union */
{
    Vehicle v;
    Land l;
    Auto a;
    Truck t;
} AnyVehicle;

typedef struct              /* the data for a all vehicles */
{
    Vtype type;
    AnyVehicle thing;
} Object;

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.

12.1.2 How polymorphism is implemented

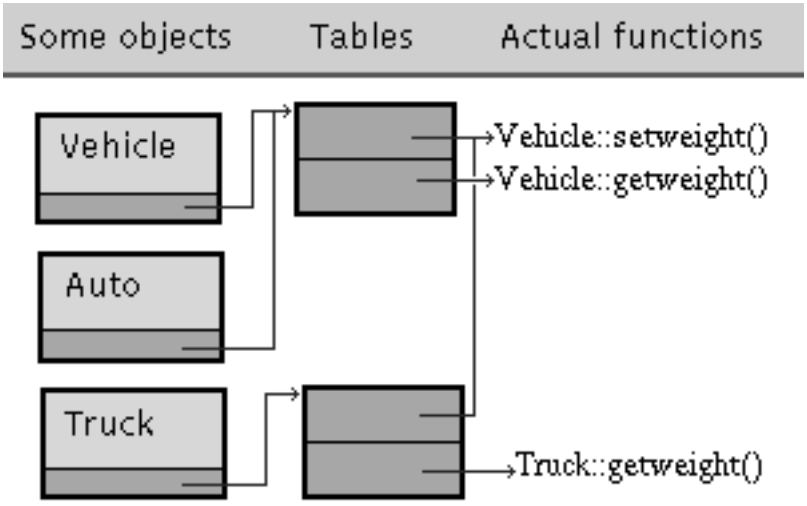
This section briefly describes how polymorphism is implemented in C++. Understanding the implementation is not necessary for the usage of this feature of C++, though it does explain why there is a cost of polymorphism in terms of memory usage.

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.



\

Figure 12.1: Internal organization objects when virtual functions are defined.

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 12.1.

As can be seen from figure 12.1, 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.

12.2 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 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 5.1), capable of comparing two `Person` objects by the alphabetical order of their names and addresses:

```
class Person: public Sortable
{
    public:
        // constructors, destructors, and stuff
        Person();
```

```
        Person(char const *nm, char const *add, char const *ph);
        Person(Person const &other);
        Person const &operator=(Person const &other);

        // 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.

12.3 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 statement

```
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.

12.4 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 virtual destructors. 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()
{
}
```

12.5 Virtual functions in multiple inheritance

As was previously mentioned in chapter 11 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:
        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 11.1). 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.

12.5.1 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`.

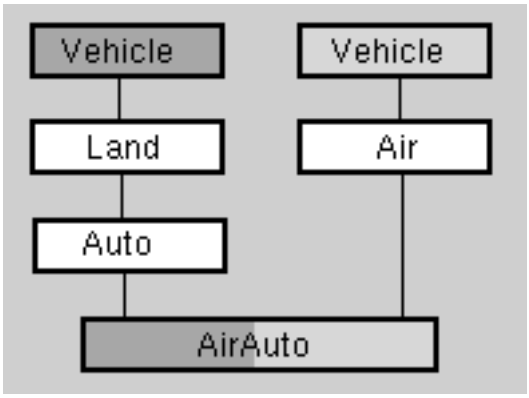
The duplication of `Vehicle` data is further illustrated in figure 12.2.

The internal organization of an `AirAuto` is shown in figure 12.3

The **C++** compiler will detect the ambiguity in an `AirAuto` object, and will therefore fail to produce code for a statement like:

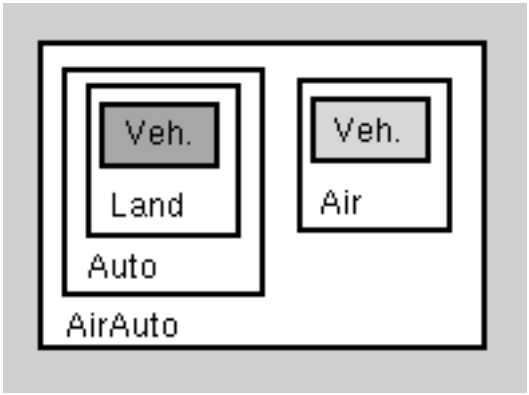
```
AirAuto
    cool;
```

[2]such as the one in James Bond vs. the Man with the Golden Gun...



\

Figure 12.2: Duplication of a base class in multiple derivation.



\

Figure 12.3: Internal organization of an `AirAuto` object.

```
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.

12.5.2 Virtual base classes

As is illustrated in figure 12.3, 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:

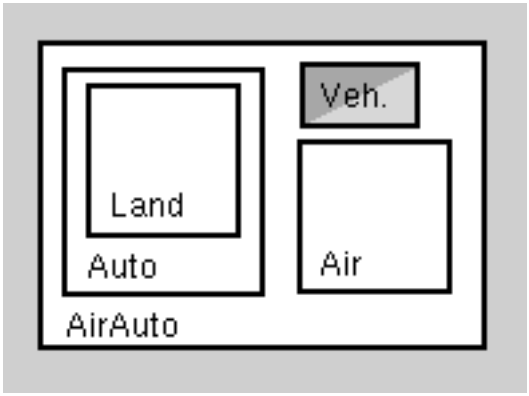
```
class Land: virtual public Vehicle
{
    ...
};

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 12.4.

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`.



\

Figure 12.4: Internal organization of an `AirAuto` object when the base classes are virtual.

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.

12.5.3 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 11.3:

```
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
        );
```

```
    }

)
```

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 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 expression-value 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
}
```

13.1.1 Compiling sources in which exceptions are used

The `Gnu g++` compiler requires a special flag to compile sources in which exceptions are used. It is quite possible that other compilers require similar flags, but that hasn't been investigated by us.

If the keywords `throw, try` or `catch` are used in a sourcetext, or if a sourcefile contains a function calling another function which may throw an exception the

```
-fhandle-exceptions
```

must be used when these sources are compiled.

The easy way-out would of course be to include the `-fhandle-exceptions` all the time, but it appears as though this doesn't always work properly, sometimes resulting in linker-problems.

Fortunately it is usually well known whether a function may throw exceptions, either directly or indirectly, and so the need for the

```
-fhandle-exceptions
```

flag is also usually well known.

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 memberfunction `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 new String("hello");  // throws a String *,
```

Although it's possible to throw objects, it is not a good idea to do so. As we have seen in section 13.2.2, objects defined locally in functions are automatically destroyed once exceptions are thrown within these functions. Consequently, a locally defined object that is thrown will also be destroyed, which is unwanted, as the object is supposed to be caught later on.

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, and in a real-life situation its destructor would normally have deleted its `name`-field. In the example, however, the memory for the name is static, and is therefore still available, so we're able to complete out illustration about what's happening when a locally defined object is thrown.

Let's first take a look at the sourcetext:

```
#include <iostream.h>

class Object
{
    public:
        Object(char const *name);
        ~Object();
        void fun();
        void hello();
    private:
        char const
            *name;
};

Object::Object(char const *n)
{
    name = n;
    cout << "Object constructor of " << name << "\n";
}

Object::~Object()
{
    cout << "Object destructor of " << name << "\n";
}

void Object::fun()
{
```

```
    Object
        toThrow("'local object'");

    cout << "Object fun() of " << name << "\n";
    throw toThrow;
}

void Object::hello()
{
    cout << "Hello by " << name << "\n";
}

int main()
{
    Object
        out("'main object'");

    try
    {
        out.fun();
    }
    catch (Object o)
    {
        cout << "Caught exception\n";
        o.hello();
    }
}
```

Now take a close look at the output generated by this program (line numbers were added by us):

```
    Object constructor of 'main object'     (1)
    Object constructor of 'local object'    (2)
    Object fun() of 'main object'           (3)
    Object destructor of 'local object'     (4)
    Caught exception                        (5)
    Hello by 'local object'                 (6)
```

```
Object destructor of 'main object'      (7)
```

The peculiarity here occurs in line (6): output generated by a local object, just before (in line (4)) destroyed by its destructor....

This is of course not what we want. But apart from that, the program behaves properly. When the exception is thrown in `Object::fun()`, the destructor of the local object `toThrow` is called. Then, the program picks up the execution again in `main()` at the `catch`-block. At that point the thrown `toThrow` object is received, and its `hello()` function is called. Fortunately, the `Object` destructor didn't destroy the name-field, otherwise the program would have tried to access foreign memory.

Summarizing, local objects should not be thrown, nor should pointers to local objects 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. So, the `Object::fun()` and the `catch`-block can be altered as follows to throw and catch pointers to `Objects`:

```
void Object::fun()
{
    cout << "Object fun() of " << name << "\n";
    throw new Object("'new object'");
}

// and in main():
...
catch (Object *o)
{
    cout << "Caught exception\n";
    o->hello();
    delete o;
}
```

Here we see that in `Object::fun()` a new `Object` is generated, which is thereupon deleted when the exception is caught.

Alternatively, realizing that references are little more than pointers, masquerading as plain variables, references could be used instead of pointers. Here's the required code:

```
void Object::fun()
{
    cout << "Object fun() of " << name << "\n";
    throw *new Object("'new object'");
}

// and in main():
...
catch (Object &o)
{
    cout << "Caught exception\n";
    o.hello();
    delete &o;
}
```

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 memberfunction `Exception::Type Exception::severity()`. This memberfunction 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 memberfunction `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 memberfunctions 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.h>

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

Templates

Most modern C++ compilers support a 'super-macro-mechanism' which allows programmers to define generic functions or classes, based on a hypothetical argument or other entity. The generic functions or classes become concrete code once their definitions are used with real entities. The generic definitions of functions or classes are called *templates*.

In this chapter we shall examine template functions and template classes.

14.1 Template functions

The definition of a template function is very similar to the definition of a concrete function, except for the fact that the arguments to the function are named in a symbolic way. This is best illustrated with an example:

```
template <class T>
void swap(T &a, T &b)
{
    T
        tmp = a;

    a = b;
    b = tmp;
}
```

In this example a template function `swap()` is defined, which acts on any type as long as variables (or objects) of that type can be assigned to each other and can be initialized by one another. The generic type which is used in the function `swap()` is called here `T`, as given in the first line of the code fragment.

The code of the function performs the following tasks:

- First, a variable of type `T` is created (this is `tmp`) and initialized with the argument `a`.
- Second, the variables which are referred to by `a` and `b` are swapped, using `tmp` as an intermediate.

The actual references `a` and `b` could refer to `int`s, `double`s or to any other type. Note that the definition of a template function is similar to a `#define` in the sense that the template function is *not yet* code, but it will result in code once it is used.

As an example of the usage of the above template function, consider the following code fragment (we use the class `Person` from section 5.1 as illustration):

```
int main()
{
    int
        a = 3,
        b = 16;
    double
        d = 3.14,
        e = 2.17;
    Person
        k("Karel", "Rietveldlaan 37", "5426044"),
        f("Frank", "Oostumerweg 17",  "4032223");

    swap(a, b);
    printf("a = %d, b = %d\n", a, b);

    swap(d, e);
    printf("d = %lf, e = %lf\n", d, e);

    swap(k, f);
    printf("k's name = %s, f's name = %s\n",
            k.getname(), f.getname());
```

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

Once the C++ compiler encounters the usage of the template function `swap()`, concrete code is generated. This means that three functions are created, one to handle `int`s, one to handle `double`s and one to handle `Person`s. The compiler generates mangled names (see also section 2.4.9) to distinguish between these functions. E.g., internally the functions may be named `swap_int_int()`, `swap_double_double()` and `swap_Person_Person()`.

It should furthermore be noted that, as far as the class `Person` is concerned, the definition of `swap()` requires a copy constructor and an overloaded assignment operator.

The fact that the compiler only generates concrete code once a template function is used, has an important consequence. The definition of a template function can never be collected in a run-time library. Rather, a template should be regarded as a kind of declaration, and should be given in a file which has a comparable function as a header file.

14.2 Template classes

The 'super-macro-mechanism' which is offered by templates can be used to define generic classes, which are intended to handle any type of entity. Typically, template classes are container classes and represent arrays, lists, stacks or trees, similar to the container classes described in chapter 15.

14.2.1 A template class: Array

As an example we present here a template class `Array`, which can be used to store arrays of any kind of element:

```
#include <stdio.h>
#include <stdlib.h>

template<class T>
class Array
```

```
{
    public:
        // constructors, destructors and such
        virtual ~Array(void)
            { delete [] data; }
        Array(int sz = 10)
            { init(sz); }
        Array(Array<T> const &other);
        Array<T> const &operator=(Array<T> const &other);

        // interface
        int size() const;
        T &operator[](int index);

    private:
        // data
        int
            n;
        T
            *data;
        // initializer
        void init(int sz);
};

template <class T>
void Array<T>::init(int sz)
{
    if (sz < 1)
    {
        fprintf(stderr, "Array: cannot create array of size < 1\n"
                        "       requested: %d\n", sz);
        exit(1);
    }
    n = sz;
    data = new T[n];
}

template <class T>
Array<T>::Array(Array<T> const &other)
{
    n = other.n;
    data = new T[n];
    for (register int i = 0; i < n; i++)
        data[i] = other.data[i];
```

```
}

template <class T>
Array<T> const &Array<T>::operator=(Array<T> const &other)
{
    if (this != &other)
    {
        delete []data;
        n = other.n;
        data = new T[n];
        for (register int i = 0; i < n; i++)
            data[i] = other.data[i];
    }
    return (*this);
}

template <class T>
int Array<T>::size() const
{
    return (n);
}

template <class T>
T &Array<T>::operator[](int index)
{
    if (index < 0 || index >= n)
    {
        fprintf(stderr, "Array: index out of bounds, must be between"
                                                    " 0 and %d\n"
                        "       requested was: %d\n",
                 n - 1, index);
        exit(1);
    }
    return (data[index]);
}
```

Concerning this definition we remark:

- The definition of the class starts with

```
template <class T>
```

This is similar to the definition of a template function: this line holds the symbolic name `T`, referring to the type which will be handled by the class.

- In the class definition, all functions which have an `Array` as their argument (e.g., the copy constructor) refer to this argument as an `Array<T>`.
- In the function definitions, the class name is referred to as `Array<T>`. The reason for this is the following: similar to name mangling in template functions, the compiler will modify the class name `Array` to a new name, when the class is concretely used. The symbolic name `T` will then become a part of the new class name.

Concerning the statements in the template we remark:

- The template class `Array` uses two data members: a pointer to an allocated array (`data`) and the size of the array (`n`).
- The class contains a copy constructor, (virtual) destructor and overloaded assignment function, since it addresses allocated memory.
- Note the statement `delete [] data` in the destructor and overloaded assignment. This statement makes sure that, when `data` points to an array of objects, the destructor for the objects is called prior to the deallocation of the array itself.
- The statement `data[i] = other.data[i]` in the overloaded assignment copies the data from another `Array`. This statement may actually copy memory byte by byte, or activate an overloaded assignment operator when the stored data is, e.g., a `Person` (see section 5.1).

Concerning the template class `Array` and in general all template classes, we have to remark that the template itself must be known to the compiler at compile-time. This usually means that the definition of the template class must be known in the *source file* in which the template is used to instantiate code. Usually this is realized by defining the template in a special header file, e.g., `array.t`[1].

By including the template header file in the source file in which the template is used, the compiler is able to create the neccesary code instantiating one or more member functions or objects of the template.

[1] note the extension `.t`, in the template defining file `array.t` which is a style-convention distinguishing *declaration header files* from *template header files*

14.2.2 Using the Array class

The template class `Array` is used as illustrated in the following example:

```
#include <stdio.h>
#include "array.t"

#define PI 3.1415

int main()
{
    Array<int>
        intarr;

    for (register int i = 0; i < intarr.size(); i++)
        intarr[i] = i << 2;

    Array<double>
        doublearr;

    for (i = 0; i < doublearr.size(); i++)
        doublearr[i] = PI * (i + 1);

    for (i = 0; i < intarr.size(); i++)
        printf("intarr[%d]   : %d\n"
                "doublearr[%d]: %g\n",
                i, intarr[i],
                i, doublearr[i]);

    return (0);
}
```

Note that the actual type of the array must be supplied when defining an object of the template class.

The class can, of course, be used with any type (or class) as long as arrays of the type can be allocated and entities of the type can be assigned. For a class such as `Person` this means that a default constructor and overloaded assignment function are needed. An illustration follows:

```
int main()
{
    Array<Person>
        staff(2);                   // array of two persons

    Person
        one,
        two;

    .                               // code assigning names and
    .                               // addresses and phone numbers
    .                               // not covered in this example

    staff[0] = one;
    staff[1] = two;

    printf("%s\n%s\n",
        staff[0].getname(), staff[1].getname());

    return (0);
}
```

Since the above array `staff` consists of `Persons`, the `Person`'s interface functions such as `getname()` can be called for elements in the array.

14.3 Templates and Exceptions

Exceptions can be used without problems in templates. Usually the exception handling is part of the normal code, while the exceptions are thrown from the template functions.

Let's take a look at the original code of the template of the `Array::init()` member, in section 14.2.1:

```
template <class T>
void Array<T>::init(int sz)
{
    if (sz < 1)
    {
```

```
        cerr << "Array: cannot create array of size < 1\n"
                        "       requested: " << sz << endl;
        exit(1);
    }
    n = sz;
    data = new T[n];
}
```

In this piece of code the request for a negative of zero sized array is punished with an `exit()`. However, if the program would be able to repair this problem elsewhere, an exception would be in order. Here's some sugggested code:

```
template <class T>
void Array<T>::init(int sz)
{
    if (sz < 1)
        throw message("Array: cannot create array of size < 1\n"
                        "       requested: %d\n", sz);
    n = sz;
    data = new T[n];
}
```

As exceptions can be used as easily within templates as they can be used outside of templates, we leave the topic of using exceptions in templates at this point, trusting that the reader will be able to generalize this example to other code.

14.4 Evaluation of template classes

In this chapter and in chapter 15 we have seen two approaches to the construction of container classes.

- The `Storable/Storage` approach from chapter 15 (see section 15.1) defines a 'storable' prototype with a pure virtual function `duplicate()`. During the storage, in the class `Storage`, this function is called to duplicate an object.

  This approach imposes the need for a duplicating function for each object which is derived from `Storable` so that it may placed in a `Storage`.

- The template approach discussed in this chapter, using the template class `Array`, poses no such restrictions when it is used. I.e., following a definition of an `Array` object, to hold say `Persons`, as in:

```
Array<Person>
    staff;
```

  the array can be used, without modifying or adapting the class `Person`.

The above comparison suggests that templates are a much better approach to container classes.

There is, however, one disadvantage: whenever a template class with a given type (`Person`, `Vehicle` or whatever) is used, the compiler must construct a *new 'real' class*, each with its own mangled name (say `ArrayPerson`, `ArrayVehicle`).

A function such as `init()`, which is defined in the template class `Array`, then occurs twice in a program: once as `ArrayPerson::init()` and once as `ArrayVehicle::init()`. Of course, this holds true not only for `init()` but for all member functions of a template class.

In contrast, the `Storable/Storage` approach from chapter 15 requires only two new functions: one duplicator for a `Person` and one for a `Vehicle`. The code of the container class itself occurs only once in a program.

Terefore, we conclude the following:

- When a program uses only one container class, the template approach is preferable: it is easier to use and requires no special precautions or conversions as far as the contained class is concerned.
- When a program uses several instances of a container class, the `Storable/Storage` approach is preferable: it prevents needless code duplication, though it does require special adaptations of the contained class.

Chapter 15

Concrete examples of C++

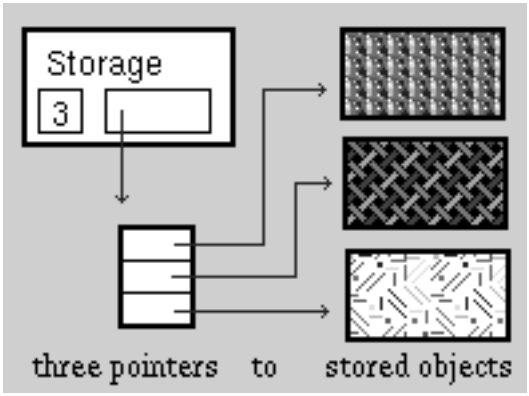
This chapter presents a number of concrete examples of programming in C++. Items from this document such as virtual functions, `static` members, etc. are rediscussed. Examples of container classes are shown.

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.

15.1 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.



\

Figure 15.1: Internal organization of the class `Storage`.

15.1.1 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 15.1.

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 15.1) 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 *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.

15.1.2 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?

  <p>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`?

  <p>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.

  <p>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 5.1. This class is re-created here, conforming to `Storable`'s protocol (only the relevant or new code is shown):

```
class Person: public Storable
{
    // 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 *(Person*)this);
}
```

15.1.3 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 15.1.1).
- 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 5.4.1.

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 *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);
}
```

15.2 A binary tree

This section shows an implementation of a binary tree in **C++**. Analogously to the classes `Storage` and `Storable` (see section 15.1) 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`.

15.2.1 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.

  <p> 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 15.1), a binary tree will contain **copies** of objects. The responsibility to duplicate an object therefore also lies with `Node`, as defined in a pure virtual function:

```
virtual Node *duplicate() 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 placed into storage in a binary tree, it can occur that the object has previously been stored. In that case the object will not be stored for a second time.

  For these cases we define a virtual function `already_stored()`, which is however not pure virtual. The default implementation will take no action. The function can however be redefined in a derived class:

  ```
  virtual void already_stored();
  ```

The complete definition and declaration of the class `Node` is given below:

```
class Node
{
    public:
        // destructor
        virtual ~Node();

        // duplicator
        virtual Node* duplicate() const = 0;

        // comparison of 2 objects
        virtual int compare(Node const *other) const = 0;
```

```
        // function to do whatever is needed to the node
        virtual void process() = 0;

        // called when object to add was already in the tree
        virtual void already_stored();
};

Node::~Node()
{
}

void Node::already_stored()
{
}
```

15.2.2 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`.

To scan a binary tree, the class `Tree` offers three methods: preorder, inorder and postorder. When scanning in preorder first a leaf in a node is processed, then the left subtree is scanned and finally the right subtree is scanned. When scanning inorder first the left subtree is scanned, then the leaf itself is processed 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:

```
class Tree
{
    public:
        // destructor, constructors
        ~Tree();
        Tree();
        Tree(Tree const &other);

        // assignment
```

```
        Tree const &operator=(Tree const &other);

        // addition of a Node
        void add(Node *what);

        // processing order in the tree
        void preorder_walk();
        void inorder_walk();
        void postorder_walk();

    private:
        // primitives
        void copy(Tree const &other);
        void destroy();

        // data
        Tree
            *left,
            *right;
        Node
            *info;
};
```

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 `copy()` and `destroy()` (these functions are presented later):

```
// destructor: destroys the tree
Tree::~Tree()
{
    destroy();
}

// default constructor: initializes to 0
Tree::Tree()
```

```
{
    left = right = 0;
    info = 0;
}

// copy constructor: initializes to contents of other object
Tree::Tree(Tree const &other)
{
    copy(other);
}

// overloaded assignment
Tree const &Tree::operator=(Tree const &other)
{
    if (this != &other)
    {
        destroy();
        copy(other);
    }
    return (*this);
}
```

Adding an object to the tree

The addition of 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 basically only a few possibilities:

- The `info` field of the current node can be a 0-pointer. In that case, a duplicate of the object to add is inserted in 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. These subtrees may first have to be allocated.
- 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. In this case, `already_stored()` is called.

The function `add()` is listed below:

```
void Tree::add(Node *what)
{
    if (! info)
        info = what->duplicate();
    else
    {
        register int
            cmp = info->compare(what);

        if (cmp < 0)
        {
            if (! left)
            {
                left = new Tree;
                left->info = what->duplicate();
            }
            else
                left->add(what);
        }
        else if (cmp > 0)
        {
            if (! right)
            {
                right = new Tree;
                right->info = what->duplicate();
            }
            else
                right->add(what);
        }
        else
            info->already_stored();
    }
}
```

Scanning the tree

The class `Tree` offers three methods of scanning a binary tree: preorder, inorder and postorder. The three functions which define these actions are recursive:

```
void Tree::preorder_walk()
{
    if (info)
        info->process();
    if (left)
        left->preorder_walk();
    if (right)
        right->preorder_walk();
}

void Tree::inorder_walk()
{
    if (left)
        left->inorder_walk();
    if (info)
        info->process();
    if (right)
        right->inorder_walk();
}

void Tree::postorder_walk()
{
    if (left)
        left->postorder_walk();
    if (right)
        right->postorder_walk();
    if (info)
        info->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.

```
void Tree::destroy()
{
    delete info;
```

```
    if (left)
        delete left;
    if (right)
        delete right;
}

void Tree::copy(Tree const &other)
{
    info = other.info ? other.info->duplicate() : 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 `copy()` is recursive. The code <tt/left = new Tree(*other.left)/ activates the copy constructor, which in turn calls `copy()` for the left branch of the tree.
- As is the case with the function `add()`, nodes themselves are duplicated with the function `duplicate()`. This function is supplied by a concrete implementation of a derived class of `Node`.

15.2.3 Using Tree and Node

We illustrate the usage of the classes `Tree` and `Node` with a program that counts words in files. Words are defined 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 `already_stored()`. This function simply increases the private counter variable `times`. Also note the use of the `new`-based function `strdupnew()`, introduced in section 5.

```
class Strnode: public Node
{
    public:
        // destructor, constructors
        ~Strnode();
        Strnode();
        Strnode(Strnode const &other);
        Strnode(char const *s);

        // assignment
        Strnode const &operator=(Strnode const &other);

        // functions required by Node protocol
        Node* duplicate() const;
        int compare(Node const *other) const;
        void process();
        void already_stored();

    private:
        // data
        char *str;
        int times;
};

Strnode::~Strnode()
{
    delete str;
}

Strnode::Strnode()
{
    str = 0;
    times = 0;
}

Strnode::Strnode(Strnode const &other)
{
    str = strdupnew(other.str);
    times = other.times;
}

Strnode::Strnode(char const *s)
{
```

```
    str = strdupnew(s);
    times = 1;
}

Strnode const &Strnode::operator=(Strnode const &other)
{
    if (this != &other)
    {
        delete str;
        str = strdupnew(other.str);
        times = other.times;
    }
    return (*this);
}

Node *Strnode::duplicate() const
{
    return (new Strnode(*this));
}

int Strnode::compare(Node const *other) const
{
    Strnode
        *otherp = (Strnode *) other;

    if (str && otherp->str)
        return (strcmp(str, otherp->str));

    if (! str && ! otherp->str)
        return (0);

    return ((int) otherp->str - (int) str );
}

void Strnode::process()
{
    if (str)
        printf("%4d\t%s\n", times, str);
}

void Strnode::already_stored()
{
    times++;
}
```

```
void countfile(FILE *inf, char const *name)
{
    Tree
        tree;
    char
        buf [255];

    while (1)
    {
        fscanf(inf, " %s", buf);
        if (feof(inf))
            break;

        Strnode
            *word = new Strnode(buf);

        tree.add(word);
        delete word;
    }
    tree.inorder_walk();
}

int main(int argc, char **argv)
{
    register int
        exitstatus = 0;

    if (argc > 1)
        for (register int i = 1; i < argc; i++)
        {
            FILE
                *inf = fopen(argv [i], "r");

            if (! inf)
            {
                fprintf(stderr, "wordc: can't open \"%s\"\n",
                         argv [i]);
                exitstatus++;
            }
            else
            {
                countfile(inf, argv [i]);
                fclose(inf);
```

```
            }
        }
        else
            countfile(stdin, "--stdin--");
    return (exitstatus);
}
```

15.3 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.

15.3.1 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 `Option`s, rather than the `Option`s 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 `Hashtable`s (the `class Hashtable` will be covered in section 15.3.6) 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);
}
```

15.3.2 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
}
```

15.3.3 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 `Objects` are stored in, e.g., `Hashtables`.

The class `Option` (cf. section 15.3.3), 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 memberfunctions 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()` memberfunction 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 `Options`. 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 memberfunction `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.

15.3.4 Derived from Option: The class TextOption

Below (in section 15.3.4) 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 15.3.3), 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);
}
```

15.3.5 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_
```

15.3.6 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 memberfunctions, 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
    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.

15.3.7 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 memberfunctions, 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 memberfunctions 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>

#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 `Hashtable`s.

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 memberfunction 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 <iostream.h>
#include <fstream.h>

#include <crux/mem.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;
};

#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_
```

15.4 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.

15.4.1 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 a `#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 `#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 memberfunctions 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++` now creates a class (`yyFlexLexer`) from which the `class Scanner` will be derived.

The code associated with the rules will be located inside the class `yyFlexLexer`. If it would be handy to access the member-functions of the derived class within

that code, it must be realized that the `this` pointer is a base-class pointer. In order to reach the derived class, the `this` pointer must be cast to a pointer to the derived class.

Looking at the rules, 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 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
------------------------------------------------------------------------ */
#include "scanner.h"        // The interface of the derived class
%}

/* ------------------------------------------------------------------------
      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 "[^>]*">"  ((Scanner *)this)->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 (!((Scanner *)this)->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 left to the derived class. 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 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. These private members are developed in practice 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 memberfunctions. They are discussed below, in the context of the memberfunctions 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);

        void switchSource();
        int  popSource();
    private:
```

```
        int const sizeof_buffer = 16384;
        int const stackDepth = 10;

        int scanYYText();   // 1: nextSource contains new name
        void performSwitch();
        void checkCircularity();
        void checkDepth();

        yy_buffer_state
            **state;
        char
            **fileName,
            *srcPtr,
            *nextSource;

        int
            stackTop;
};
```

The `switchSource()` memberfunction 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 memberfunction `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 memberfunctions 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 `stackDetph` is reached, the stack is full, and the program aborts with an appropriate message. For this the memberfunction `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 memberfunction `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 memberfunction `yy_switch_to_buffer`.

The sources for the memberfunctions `performSwitch()`, `checkDepth()`, and `checkCircularity()` is 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 memberfunction `popSource()` is called to pop the previously pushed source-file 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 `yyFlexLexer` part of the `Scanner` object. The `Scanner` object itself only has two public memberfunctions: one to push a sourcefile on a stack when a switch to the next sourcefile is requested, the other one restores the previously pushed source.

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);

    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
  ```

15.4.2 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 bison++ specification file

The bison specification file as used with `bison` is comparable to the specification file as 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 contains 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. However, several extra 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. Normally the definition of the body of the lexer-macro as called by the parser must be defined, and normally the body of the error-macro must be defined. Specifically, the following symbols are recognized by `bison++`, and can be redefined in this section:

  - `%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();
    ```

  - `%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 will contain the code 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 can be defined here. It can be defined as `= 0`, in which case the parser's class will again become abstract. Otherwise, it can be used to specify the inner workings of the error function. E.g.,

  ```
  %define ERROR_BODY { cerr << "syntax Error\n"; }
  ```

- 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.
  - `%define CONSTRUCTOR_INIT :initializer(s)`: this defines the base-class and member initializers for the constructor.
  - `%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;
  };
  ```

- Associating tokens and unionfields. Tokens can be associated with unionfields. By doing so, the parser's actions-code becomes much cleaner than if the tokens aren't associated with fields. Moreover, nonterminals can also be associated with unionfields. In these cases 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. In order to associate a nonterminal or a token with a unionfield, the `<fieldname>` specification is used. E.g.,

```
%token <i> INT
%token <d> DOUBLE
%type  <i> intExpr
```

In this example, note that both the tokens and the nonterminals can be associated with a field of the union. This will be further illustrated in the upcoming description of the rules of the grammar.

- 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
;

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
;
```

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  <i> INT
%token  <d> DOUBLE
%type   <i> intExpr
%type   <d> doubleExpr

%left   '+'
%left   '*'
%right  UnaryMinus

%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.

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 <iostream.h>
#include "parser.h"

extern yyFlexLexer
    lexer;
extern Parser
    parser;
%}
%%
[ \t]                       ;
[0-9]+                      {
                                parser.yylval.i = atoi(yytext);
                                return(Parser::INT);
                            }
"."[0-9]*                   |
[0-9]+("."[0-9]*)?          {
                                parser.yylval.d = atof(yytext);
                                return(Parser::DOUBLE);
                            }
.|\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"
Parser
        parser;
yyFlexLexer
    lexer;
int main()
{
    return (parser.yyparse());
}
```