C++ Annotations
Version 3.4.13

Frank B. Brokken and Karel Kubat
(frank@icce.rug.nl, karel@icce.rug.nl)
ICCE, State University of Groningen
Westerhaven 16, 9718 AW Groningen
Netherlands
Published at the State University of Groningen
ISBN 90 367 0470 7 1994, 1995

This document is intended for knowledgable users of C who would like to make the transition to C++. It is a guide for 'our' programming courses, which are given yearly at the State University of Groningen. As such, this document is not a complete C/C++ handbook, but rather serves as an addition to other documentation sources.

Contents

1 Introduction to C++

This document presents an introduction to programming in **C++**. It is a guide for 'our' programming courses, which are given yearly at the State 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 an 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 3.4.13) is updated when the contents of the document are changed. 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.

This document is published by the ICCE, State University of Groningen, the Netherlands. Original Dutch version by Frank B. Brokken and Karel Kubat. English version, editing, typesetting, extensions to the text and printing by Karel Kubat. This document was typeset using the `linuxdoc-sgml` formatting system.

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

1.1 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.

[1] e.g., the Dutch book *Werken met C*, Brokken and Kubat, Sybex 1991, or the Microsoft **C/C++** tutorial

1.1.1 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, while an `int` is still an `int`. As we shall see later (see section 1.4.10), 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.

1.2 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 6).

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 worth-while investigating.

1.3 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`).

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

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

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

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

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

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

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

The hierarchy of the thus identified objects further illustrated in figure 1.

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

1.4 Differences between C and C++

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

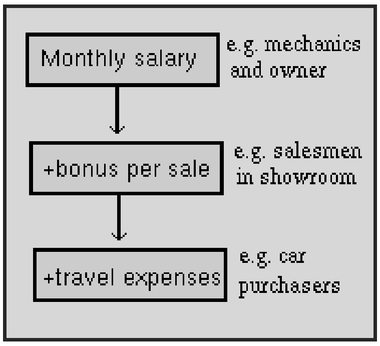

Figure 1: Hierarchy of objects in the salary administration.

1.4.1 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 in some **C** compilers, such as the Microsoft **C** Compiler V5.

1.4.2 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.

1.4.3 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.

1.4.4 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.

1.4.5 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.

1.4.6 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:

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

```
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 often used method to declare external **C** functions is presented below.

1.4.7 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 made 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.

1.4.8 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. The variable does not exist prior to the statement. In some compilers, the variable continues to exist after the execution of the `for` statement (notably, in the `g++` compiler V2.6) while in other compilers the variable can only be used inside the `for`.

Defining local variables at any position in the code can lead to less readable code. Two hints are:

- 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.

1.4.9 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 ()
{
    register int
        counter;                    // local variable

    for (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.

1.4.10 Function Overloading

In **C++** it is possible to define several functions with the same name but which perform different actions. The functions must only differ in the argument list. 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.

1.4.11 Default function arguments

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

An example is shown below:

```
#include <stdio.h>

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

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

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

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

Functions may be defined with more than one default argument:

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

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

```
}

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

    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.

1.4.12 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`, `enum` 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;
```

1.4.13 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 1.3), 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 2.2).
- 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.

2 A first impression of C++

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

2.1 More extensions of C++

Before we continue with the 'real' object-oriented approach to programming, we first introduce further extensions of **C++**.

2.1.1 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 << and >>. This is illustrated in the example below:

```
#include <iostream.h>

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

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

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

    return (0);
}
```

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) which processes the input and output of a program. Note that the terminology 'object' means here: the set of data and functions which define the item in question.
- The stream `cin` reads data and copies the information to variables (e.g., `ival` in the above example) using the right-shift operator >>. We will describe later how operators in **C++** can perform quite other actions than what they are defined to do by the language grammar, such as is the case here.
- The operators which manipulate `cin`, `cout` and `cerr` (which are >> and <<) also manipulate variables of different types. In the above example `cout << ival` stands for the printing of an integer value. The actions of the operators therefore depend on the type of supplied variables.

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 operators << and >> with their respective streams 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 (usually) doesn't warn. In contrast, argument checking with `cin`, `cout` and `cerr` is performed by the compiler.
- 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++**, but in our opinion hardly improves the readability of programs (rather the opposite is true).

2.1.2 The keyword const

The keyword `const` very often occurs in **C++** programs, even though it is also part of the **C** grammar.

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.

2.1.3 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 role 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 naturally also be used in **C++** but the same effect can be achieved with 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 implements references is by internally 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.[5]

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

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

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

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

int main ()
{
    int
        x;

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

- When a function changes the value of its argument, the address can be passed:

```
void increase (int *valp)
{
```

[5]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.

```
    *valp += 5;
}

int main ()
{
    int
        x;

    increase (&x);              // a pointer is passed, so
    return (0);                 // x might be changed
}
```

- References can be reserved for those cases where the argument will not be changed by the function, but where it is desirable to pass a reference (i.e., in fact a pointer) 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: in this case the stack usage becomes significant when a copy of the entire structure is passed.

  This is illustrated below:

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

void printperson (person &p)    // printperson expects a
{                               // reference to a structure
    printf ("Name: %s\n"
            "Address: %s\n",
        p.name, p.address);
}

int main ()
{
    person
        boss;

    .
    .
    printperson (boss);             // no pointer is passed,
}                                   // so variable won't be
                                    // altered by function
```

An improvement is the addition of the keyword `const`:

```
void printperson (person const &p)
```

After such a declaration of the function `printperson()` no ambiguity exists.

- (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 4.)

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 (void)
{
    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 (however, there are situations where it is useful to return a reference, this is discussed later). **C++** provides as far as references are concerned a powerful tool, which however can mutilate the code.

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? An exception to this rule are references which are declared as `external` (these references are assumed to be initialized elsewhere), references as arguments of functions (the caller then initializes the reference), references as return types of functions (the function determines to what the return value will refer) and references as data members of classes (we will describe this later).

  In contrast, pointers are variables by themselves. They point, whether at something concrete or just "at nothing".

- Except for the case of references as return values of functions, 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 on 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.

2.2 Functions as part of structs

The first chapter described that functions can be part of `struct`s (see section 1.4.13). Such functions are called 'member' functions. 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**.

2.3 Data hiding: public, private and class

As mentioned previously (see section 1.2), **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)[6]. 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
```

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

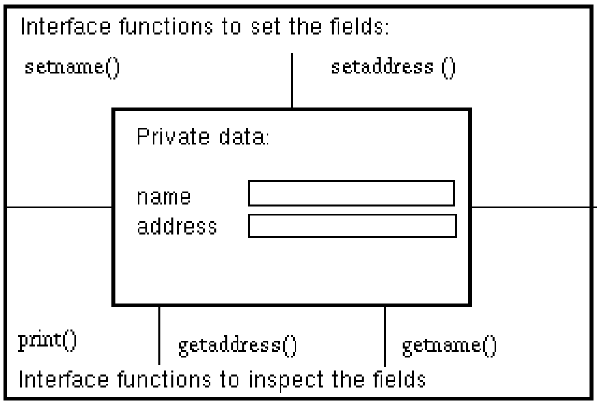

Figure 2: Private data and public interface functions of the class Person.

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 further illustrated in figure 2.

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 ( (char const *) 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`).

2.4 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 `struct`s are concerned. In **C** it is not uncommon 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");
```

is in **C++**:

```
Person
    x;

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

3 Classes

The usage of classes is further explained in this chapter. Two special member functions, the constructors 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.

The definition of a `Person` so far is as follows:

```
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 the class; either to set the data fields to a given value (e.g., `setname()`) or to inspect the data (e.g., `getname()`).

3.1 Constructors and destructors

A class in **C++** may contain two special functions which are involved in the internal workings of the class. These functions are the constructors and the destructor.

3.1.1 The constructor

The constructor 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., which performs no actions) when a corresponding object is created. The actual generated code of course depends on the compiler.[7]

When an object is a local non-static variable in a function, the constructor is called when the function is executed. When an object is a global or a static variable, the constructor is called when the program starts; even before `main()` gets executed.

This is illustrated in the following listing:

```
#include <stdio.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
```

[7]A compiler-supplied constructor in a class which contains composed objects (see section 3.5) will 'automatically' call the member initializers, and therefore does perform some actions. We postpone the discussion of such constructors to 3.5.1.

```
    g;                      // global object

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

    puts ("here's function func()");
}

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

    puts ("main() function");
    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 remark the following:

- 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 ()
  {
      .
      .
      .
  }
  ```

- The constructor function in the example above has no arguments; it is therefore also called the **default** constructor. This is however no requirement *per sé*. 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 for illustratory 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()
```

3.1.2 The destructor

The second special 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 function in which the object is defined is about to return; for static or global variables the destructor is called before the program terminates. Even when a program is interrupted using an `exit()` call, the destructors are called for objects which exist at that time.

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.

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

```
class Test
{
    public:
        Test ();                        // constructor
        ~Test ();                       // destructor
        .
        .
};
```

3.1.3 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.

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

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

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

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 address 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.

The interface functions `get...()` are listed below:

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

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

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

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

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

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

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

To demonstrate the usage of the class `Person`, a code example follows below. 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, instead of 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`.

```
void printperson (Person const &p)
{
    printf ("Name    : %s\n"
            "Address : %s\n"
            "Phone   : %s\n",
    p.getname (), p.getaddress (), p.getphone ());
}

int main ()
{
    Person
        p;

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

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

At this point it we should also note that the above code fragment can only serve as an example: most **C++** compilers will actually fail to parse the code. The reason for this is that the function `printperson()` receives a **const** argument, but calls functions for this argument which might or might not modify it (these are the functions **getname()**, **getaddress()** and **getphone()**). Given this setup, the 'constness' of the argument to **printperson()** cannot be guaranteed – and hence, the compiler will not produce working code. The solution would of course be to tell the compiler that **getname()**, **getaddress()** and **getphone()** will not modify the object at hand: but we postpone this modification to section 3.2.

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 **printf()**. This anesthetic feature can be remedied with a little more code:

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

3.1.4 Constructors with arguments

In the above definition of the class **Person** the constructor and destructor have no arguments. **C++** allows the constructor to be defined with an argument list which is 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 definition. A declaration in, e.g., a header file, would then look as follows:

```
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", "426044"),
        b;

    .
    .
}
```

The order of construction The possibility to define arguments with 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 below 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 = strdup ("without name");
    printf ("Test object without name created\n");
}

Test::Test (char const *name)
{
    n = strdup (name);
    printf ("Test object %s created\n", n);
}

Test::~Test ()
{
```

```
    printf ("Test object %s destroyed\n", n);
    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 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
```

3.2 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 could 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. Again the rule of thumb from section 2.1.2 applies: whichever appears **before** the keyword `const`, may not be altered or doesn't alter data.

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

```
    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 lead to a compilation error.

The purpose of `const` functions lies in the fact that **C++** allows `const` objects to be created. For such objects only the functions which do not modify it; i.e., the `const` member functions, may be called. The only exception to the rule are the constructor and destructor: these are called 'automatically'. This feature is comparable to the definition of a variable `int const max = 10`: such a variable may be **initialized** at its definition. Analogously the constructor can **initialize** its object at the definition, but subsequent assignments may not take place.

The following example shows how a `const` objects of the class `Person` can be defined. At the definition of an object the data fields are initialized (this is an action of the constructor):

```
    Person
        const me ("Karel", "karel@icce.rug.nl", "426044");
```

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.

Generally it is a good habit to declare member functions which do not modify their object as `const`. This subsequently allows the definition of `const` objects.

3.3 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 memory to which is allocated by `new`. This memory is later released by the operator `delete`.

```
int
    *ip;

ip = new int;
.
.
delete ip;
```

Note that `new` and `delete` are operators and therefore do not require parentheses, such as is the case with functions like `malloc()` and `free()`.

3.3.1 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 the 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 is here: whenever `new` is followed by `[]`, `delete` should be followed by it too.

3.3.2 New and delete and object pointers

The operators `new` and `delete` are also used when an object of a given class is allocated. The advantage of the operators over functions as `malloc()` and `free()` lies in the fact that `new` and `delete` call the corresponding constructor or destructor. This is illustrated in the below 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", "05903-2223");
.
.
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 which the object uses. 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. To release such an array, the array operators `[]` must be used with the `delete` operator:

```
delete [] personarray;
```

The presence of the `[]` ensures that the destructor is called for each object in the array. Note that `delete personarray` would only release the memory of the array itself.

3.3.3 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[8].
- 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 argument (a `void` argument list),
- returns no value (`void` return type).

The redefined error function might, e.g., print a message and terminate the program. The error function is included in the allocation system by the function `set_new_handler()`, defined in the header file `new.h`. On some compilers, notably the Microsoft **C/C++** 7 compiler, 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 Microsoft **C/C++** requirements:

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

int out_of_memory (size_t sz)
{
    printf ("Memory exhausted, can't allocate %u bytes\n", sz);
    exit (1);

    return (0);     // return an int to satisfy the
                    // declaration
}

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

    // install error function
    set_new_handler (out_of_memory);

    // eat up all memory
    puts ("Ok, allocating..");
    while (1)
    {
```

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

```
            ip = new int [100];
            total_allocated += 100L;
            printf ("Now got a total of %ld bytes\n",
                    total_allocated);
        }

        return (0);
    }
```

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

Often, even standard **C** functions which allocate memory, such as **strdup()**, **malloc()**, **realloc()** etc. trigger the **new** handler when memory allocation fails. This means that once a **new** handler is installed, such functions can be used in a **C++** program without testing for errors. However compilers exist where the **C** functions do not trigger the **new** handler.

3.4 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 23", "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 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 with `inline` functions, which can be defined in two ways.

3.4.1 Inline functions within class definitions

Using the first method to implement `inline` functions, the code of a function is defined **in a class definition 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 definition of the class `Person`. The keyword `const` occurs after the function declaration, and before the code block.

The effect of this is the following. When `getname()` is called in a program statement, the compiler generates the code of the function instead of a call. This construction is called an inline function because the compiler as it were 'inserts' the actual code of the function.

3.4.2 Inline functions outside of class definitions

The second way to implement inline functions leaves a class definition intact, but mentions the keyword `inline` in the function definition. The class and function definitions then are:

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

inline char const *Person::getname () const
```

```
{
    return (name);
}
```

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

3.4.3 When to use inline functions

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

- In general `inline` functions should **not** be used. *Voilà.*
- 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
  {
      puts (name);
  }
  ```

  This function, which is presumed to be a member of the class `Person` for the sake of the argument, contains only one statement: but a statement which takes relatively a long time to execute. In general, functions which perform input and output spend lots of time. The effect of the conversion of this function `printname()` to `inline` would therefore lead to unmeasurable time gain.

`inline` functions have one disadvantage: the actual code is inserted by the compiler and must therefore be known compile-time. Therefore an `inline` function can never be located in a run-time library. Practically this means that an `inline` function is placed near the definition 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**.

3.5 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 include information about the birth date:

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

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

        char const *getname () const;
        char const *getaddress () const;
        char const *getphone () const;
        int getbirthyear () const;
        int getbirthmonth () const;
        int getbirthday () const;

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

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

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

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

Composition is not extraordinary or **C++** specific: in **C** it is quite common to include `struct`s or `union`s in other compound types.

3.5.1 Composition and const objects: member initializers

Composition of objects has an important consequence for the constructor functions of the 'composed' (embedded) object. Unless explicitely 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 = strdup (nm);
    address = strdup (adr);
    phone = strdup (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. 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 by 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** be defined in the composed class.

Not using member initializers can also lead to inefficient code. E.g., 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 = strdup (nm);
    address = strdup (adr);
    phone = strdup (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 may even not work when the composed object is declared as `const`. A data field like `birthday` is a good candidate for being `const`, since a person's birthday is not likely to change.

This means that when the definition of a `Person` is changed so that the data member `birthday` is declared as `const`, the implementation of the constructor `Person::Person()` with six arguments **must** use member initializers. The call to `birthday.set...()` would be illegal, since this is no `const` function.

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 leads to more efficient code, but also allows the composed object to be declared as `const`.

3.6 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 function `decrement()` and the function `touch()` of the class `B` attempt to access a `private` datamember of `A`.

We can explicitely allow `decrement()` to access `A`'s data, and we can explicitely 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 creates exceptions to the rule of data hiding. Using friend classes should therefore be minimized to those cases where it is absolutely essential.

4 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.

The comparison between `free()` and `delete` is analogous: `delete` makes sure that when an object is deallocated, a corresponding destructor is called.

The calling of constructors and destructors when objects are created or destroyed, has a number of consequences which shall be discussed in this chapter. Many problems in program development in **C** 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. In this chapter the following topics are discussed:

- the assignment operator and operator overloading,
- the `this` pointer,
- the copy constructor.

4.1 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;
}

void main ()
{
    Person
        kk ("Karel", "Rietveldlaan",
            "050-426044"),
        *bill = new Person ("Bill Clinton",
               "White House",
               "09-1-202-142-3045")

    printf("%s, %s, %s\n",
        kk.getname (), kk.getaddress (), kk.getphone ());
    printf("%s, %s, %s\n",
        bill->getname (), bill->getaddress (), bill->getphone ());

    delete bill;
}
```

The memory which is occupied by the object **`kk`** is released automatically when **`main()`** terminates: the **C++** compiler makes sure that the destructor is called. Note however 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.

4.2 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 byte-by-byte copying from one compound type to the other.

Let us now consider the consequences of this default action in a program statement as the following:

```
void printperson (Person const &p)
{
    Person
        tmp;

    tmp = p;
    printf ("Name:     %s\n"
            "Address:  %s\n"
```

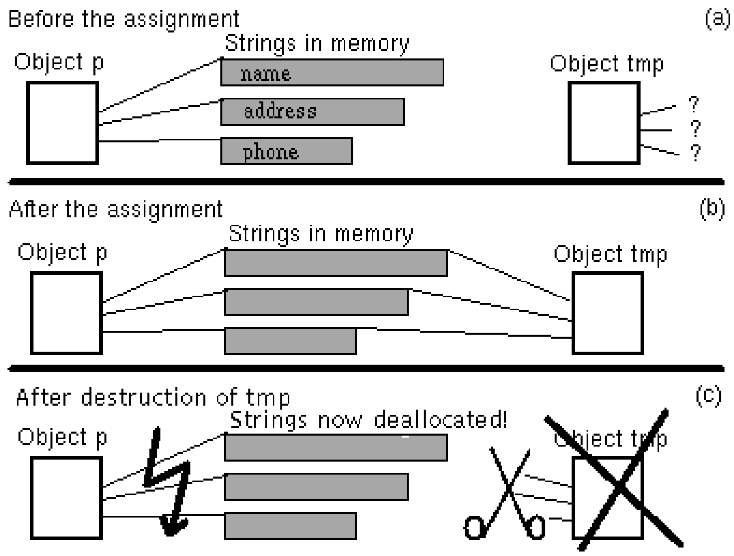

Figure 3: Private data and public interface functions of the class Person, when assigning byte by byte.

```
                "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 3.

After the execution of `printperson()` , the object which was referenced by `p` may still contain valid pointers to strings, but pointers which address deallocated memory. This action is undoubtedly not a desired effect of

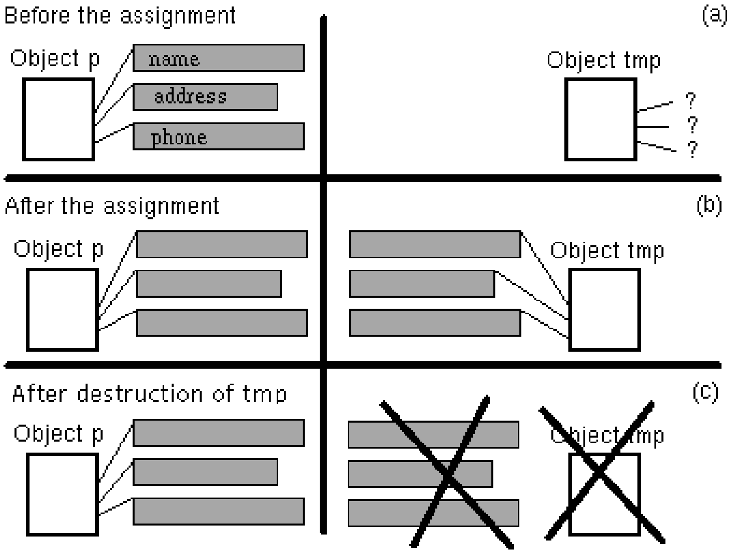

Figure 4: Private data and public interface functions of the class Person, when using the 'correct' assignment.

a function like the above. The deallocated memory will likely become occupied during subsequent allocations, thereby causing the previously held strings to become lost.

In general it can be concluded that **every class which contains a constructor and a destructor, and which contains pointer fields to address allocated memory, is a potential candidate for trouble**. There is of course a possibility to intervene: this possibility will be discussed in the next section.

4.2.1 Overloading the assignment operator

Obviously, the right way to assign one `Person` object to another, is **not** to copy the contents of the object byte by byte. 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 4.

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's data
    name = strdup (other.name);
```

```
        address = strdup (other.address);
        phone = strdup (other.phone);
    }
```

Using this tool we could rewrite the offending function `func()`:

```
    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 purely symptomatic. 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. Our experience shows that *errare humanum est*; a solution which doesn't enforce exceptions is therefore preferable.

The problem of the assignment operator is solved by using **operator overloading**: the syntactic possibility of **C++** to redefine the actions of an operator in a given context. Operator overloading was discussed earlier, when the operators << and >> were redefined for the usage with streams as `cin`, `cout` and `cerr` (see section 2.1.1).

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 << and >> in the context of streams are misleading: the stream operations do not have anything in common with the bitwise shift operations.

The function operator=() To achieve operator overloading in a context of a class, the class is simply expanded with a `public` function which states the operator. A corresponding function is then defined.

For example, to overload the addition operator `+` a function `operator+()` would be defined. The function name consists of the keyword `operator` and 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
void Person::operator= (Person const &other)
{
    // deallocate old data
    delete name;
    delete address;
    delete phone;

    // make duplicates of other's data
    name = strdup (other.name);
    address = strdup (other.address);
    phone = strdup (other.phone);
}
```

The function `operator=()` which is presented above is the first version of the overloaded assignment. We shall present better and less bug-prone versions later.

The actions of this member function are similar to those of the previously proposed function `assign()`, but the name makes sure that this function is also activated when the assignment operator `=` is used. In fact there are two ways to call this function, which are illustrated below:

```
Person
    pers ("Frank", "Oostumerweg 23", "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. The code fragment however illustrates the similarity of the two methods of calling the function.

4.3 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[9]. 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 = strdup (n);
}

// alternative 2: explicit usage of this
void Person::setname (char const *n)
{
    delete this->name;
    this->name = strdup (n);
}
```

Explicit usage of the `this` pointer is not very frequent. There is however a number of situations where the `this` pointer is needed.

[9] `this` is not available in the not yet discussed `static` member functions

4.3.1 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 function 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 assigned are first released, but this also leads to the releasing of the strings of the right-hand side variable! 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. Auto-assignment can however occur in more hidden forms:

```
Person
    one,
    two,
    *pp;

pp = &one;
.
.
*pp = two;
.
.
one = *pp;
```

The problem of the auto-assignment can be solved by 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 = strdup (other.name);
            address = strdup (other.address);
            phone = strdup (other.phone);
        }
    }
```

This is the second version of the overloaded assignment function. One, yet better version remains to be discussed.

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`.

4.3.2 Associativity of operators and this

The syntax of **C++** states that the associativity of the assignment operator is to the right-hand side; i.e., in a statement as

```
    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 not quite syntactically right.

The syntactical problem can be illustrated as follows. When we rewrite the expression `a = b = c` to the form which explicitly mentions member functions, we get:

```
    a.operator= (b.operator= (c));
```

This 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 by using the `this` pointer. The overloaded assignment function expects as its argument a reference to a `Person` object; in the same way it can 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 (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 = strdup (other.address);
        name = strdup (other.name);
        phone = strdup (other.phone);
    }

    // return current object, compiler will make sure
    // that a const reference is returned
    return (*this);
}
```

4.4 The copy constructor: Initialization vs. assignment

In the following sections we shall look closer at another usage of the operator `=`. We shall use a class `String` as an example. This class is meant to handle allocated strings and is defined as follows:

```
class String
{
    public:
        // constructor, destructors
        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 definition we remark the following:

- The class contains a pointer `char *str` to address allocated memory. For this reason, the class has a constructor and 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 = strdup (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 below 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 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 code fragment uses the operator `=`, this is no **assignment**: rather an **initialization**, and hence, construction.

- In statement 2 a second `String` object is created. Again a constructor is called; but since no special arguments are present, this is the default constructor.

- In statement 3 a new object `c` is created. Again, a constructor is therefore called. The new object is also initialized with (the data of) object `a`.

  This form of initializations has not been discussed yet. Since we can rewrite this statement in the form

  ```
  String
      c (a);
  ```

  this initialization 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.

- In statement 4 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 which applies here is that **whenever an object is created, a constructor is needed**. The form of the constructor is still the following:

- The constructor has no return value.
- The constructor is defined in a function with the same name as the corresponding class.
- The argument list of the constructor can be deduced from the code; the argument is either present between parentheses or following a `=`.

We conclude therefore that, given the above code 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 = strdup (other.str);
}
```

The actions of the copy constructor are similar to those of the overloaded assignment operator function: an object is **duplicated**, so that it contains its own allocated data. The copy constructor function is however 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, this 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; 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; 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 by defining `func()` with 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`: this is indeed possible, given a constructor which expects 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.

4.4.1 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. We remark that:

- The duplication of (private) data occurs in the copy constructor and in the overloaded assignment function.
- The deallocation of used memory occurs in the overloaded assignment function and in the destructor.

The two above actions (duplication and deallocation) can be coded in two primitive 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 **unconditionally** copy the data of another object or to deallocate the memory of the current object. 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 implementation of the functions `copy()` and `destroy()`:

```
// copy(): unconditionally copy other object's data
void Person::copy (Person const &other)
{
    name = strdup (other.name);
    address = strdup (other.address);
```

```
    phone = strdup (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);
}
```

4.5 More operator overloading

The following sections present more examples of operator overloading.

4.5.1 Overloading operator[]

As one more example of operator overloading, we present here a class which is meant to represent an array of `int`s. Indexing the array elements occurs with the standard array operator `[]`, but additionally the class checks for boundary overflow.

An example of the usage of the class is given below:

```
int main ()
{
    Intarray
        x (20);             // 20 ints

    for (register int i = 0; i < 20; i++)
        x [i] = i * 2;      // assign the elements

    for (i = 0; i <= 20; i++)
        printf ("At index %d: value %d\n",
                i, x [i]);

    return (0);
}
```

This example shows how an array is created to hold 20 `int`s. The elements of the array can be assigned or retrieved. The above example should produce a run-time error, which is generated by the class `Intarray`: the last `for` loop causes a boundary overflow, since `x[20]` is addressed while legal indexes are range from 0 to 19.

The definition of the class is given below:

```
class Intarray
{
     public:
        // constructors, destructor etc.
        Intarray (int sz = 1);          // default size: 1 int
        Intarray (Intarray const &other);
        ~Intarray ();
        Intarray const &operator= (Intarray const &other);

        // the interface
        int &operator[] (int index);

    private:
        // data
        int *data, size;
};
```

Concerning this class definition 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 uses internally a pointer to address allocated memory. Hence, the necessary tools are provided: a copy constructor, an overloaded assignment function and a destructor.
- The interface is defined as a function which returns a reference to an **int**. This allows an expression like `x[10]` to be used 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 set data of the class.

The member functions of the class are given below.

```
// constructor
Intarray::Intarray (int sz)
{
    // check for legal size specification
    if (sz < 1)
    {
        printf ("Intarray: size of array must be >= 1, not %d!\n", sz);
        exit (1);
    }

    // remember size, create array
    size = sz;
    data = new int [sz];
}

// copy constructor
Intarray::Intarray (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];
}

// overloaded assignment
Intarray const &Intarray::operator= (Intarray const &other)
{
    // take action only when no auto-assignment
    if (this != &other)
    {
```

```
            // set size
            size = other.size;
            // remove previous memory, create new array
            delete [] data;
            data = new int [size];
            // copy other's data
            for (register int i = 0; i < size; i++)
                data [i] = other.data [i];
        }
        return (*this);
    }

    // here is the interface function
    int &Intarray::operator[] (int index)
    {
        // check for array boundary over/underflow
        if (index < 0 || index >= size)
        {
            printf ("Intarray: boundary overflow or underflow, "
                    "index=%d, should range from 0 to %d\n",
                    index, size - 1);
            exit (1);
        }

        // emit the reference
        return (data [index]);
    }
```

4.5.2 Cin, cout, cerr their operators

This section describes how a class can be adapted for the usage with the **C++** streams `cout` and `cerr` and the operator <<. Adaptation of a class for the usage with `cin` and its 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 section 4.1) , 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
extern 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.

  An improved form of the above function would therefore be:

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

4.6 Conclusion

Two important extensions to classes have been discussed in this chapter: the overloaded assignment operator and the copy constructor. As we have seen, classes with pointer data which address allocated memory are potential sources of semantic errors. The two introduced extensions are the only measures against unintentional loss of allocated data.

The conclusion is therefore: as soon as a class is defined where pointer data are used, an overloaded assignment function and a copy constructor should be implemented.

5 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 would then 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++ therefore allows `static` data and functions, which are common to all objects of a class. These functions and data will be discussed in this chapter.

5.1 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 for each object of the class. A `static` data member is created when the program starts executing; but is nevertheless part of the class.

`static` data members which are declared `public` are like 'normal' global variables: they can be addressed by all code of the program by using their name, preceded by the class name and by 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 **declaration**, and not the **definition** of `static` data members. We will discuss the definition of such members shortly.

5.1.1 Private static data

To illustrate the usage of a `static` data member which is a `private` variable in a class, consider the following code fragment:

```
class Directory
{
    public:
        // constructors, destructors, etc. are
        // not shown here
        .
        .
    private:
        // data members
        static char path [];
};
```

The data member `path` is a `private static` variable. It exists only once, 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 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. The data member `path` from the above class `Directory` could thus be defined and initialized in the source file of the constructor:

```
// 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 occurs only once. During the class definition the `static` member is actually only **declared**; during its definition the type and class name are explicitly stated. Note also that the size specification can be left out of the declaration, as is the case in the above array `path`; it is however needed in the definition.

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 definition is given below:

```
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. When the last object is destroyed, 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.

5.1.2 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 section 5.1 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: the class definition would still contain only the declaration of the string `path`. This means that some source file would still need to contain the definition

```
char
    Directory::path [200];
```

5.2 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` part of a class definition 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 [];
};

// definition 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 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.

6 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 3.5), 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

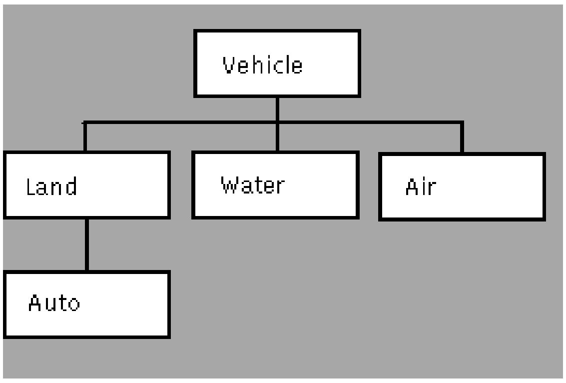

Figure 5: Initial object hierarchy of vehicles.

be **defined** by means of an older, pre-existing, class; which leads to a situation in which a new class has all the functionality of the older class, and additionally introduces its own specific functionality. Instead of composition, where a given class **contains** another class, we mean here derivation, where a given class **is** another class.

Another term for derivation is **inheritance**: the new class inherits the functionality of an existing class, while the existing class does not appear as a data member in the definition of the new class. When speaking of inheritance the existing class is called the **base class**, while the new class is called the **derived class**.

Derivation of classes is often used when the methodology of **C++** program development is fully exploited. In this chapter we will first address the syntactical possibilities which **C++** offers to derive classes from other classes. Then we will address the peculiar extension to **C** which is thus offered by **C++**.

As we have seen the object-oriented approach to problem solving in the introductory chapter (see section 1.3), 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 5.

6.1 Related types

The relationship between the proposed classes representing different kinds of vehicles is now further illustrated. The figure shows the object hierarchy in vertical direction: an `Auto` is a special case of a `Land` vehicle, which in turn is a special case of a `Vehicle`.

The class `Vehicle` is thus the 'greatest common denominator' in the classification system. For the sake of the example we implement in this class the functionality to store and retrieve the weight of a vehicle:

```
class Vehicle
{
    public:
        // constructors
        Vehicle ();
        Vehicle (int wt);

        // interface
        int getweight () const;
        void setweight (int wt);

    private:
        // data
        int weight;
}
```

Using this class, the weight of a vehicle can be defined as soon as the corresponding object is created. At a later stage the weight can be re-defined or retrieved.

To represent vehicles which travel over land, a new class `Land` can be defined with the functionality of a `Vehicle`, but in addition its own specific information. For the sake of the example we assume that we are interested in the speed of land vehicles **and** in their weight. The relationship between `Vehicle`s and `Land`s could of course be represented with composition, but that would be awkward: composition would suggest that a `Land` vehicle **contains** a vehicle, while the relationship should be that the `Land` vehicle **is** a special case of a vehicle.

A relationship in terms of composition would also introduce needless code. E.g., consider the following code fragment which shows a class `Land` using composition (only the `setweight()` functionality is shown):

```
class Land
{
    public:
        void setweight (int wt);
    private:
        Vehicle v;      // composed Vehicle
};

void Land::setweight (int wt)
{
    v.setweight (wt);
}
```

Using composition, the `setweight()` function of the class `Land` would only serve to pass its argument to `Vehicle::setweight()`. Thus, as far as weight handling is concerned, `Land::setweight()` would introduce no extra functionality, just extra code. Clearly this code duplication is redundant: a `Land` should **be** a `Vehicle`, and not: a `Land` should **contain** a `Vehicle`.

The relationship is better achieved with inheritance: `Land` is **derived** from `Vehicle`, in which `Vehicle` is the base class of the derivation.

```
class Land: public Vehicle
{
    public:
        // constructors
        Land ();
        Land (int wt, int sp);

        // interface
        void setspeed (int sp);
        int getspeed () const;

    private:
        // data
        int speed;
};
```

By postfixing the class name **Land** in its definition by **public Vehicle** the derivation is defined: the class **Land** now contains all the functionality of its base class **Vehicle** plus its own specific information. The extra functionality consists here of a constructor with two arguments and interface functions to access the **speed** data member.[10]

To illustrate the usage of the derived class **Land** consider the following example:

```
Land
    veh (1200, 145);

int  main ()
{
    printf ("Vehicle weighs %d\n"
            "Speed is %d\n",
        veh.getweight (), veh.getspeed ());
    return (0);
}
```

This example shows two features of derivation. First, **getweight()** is no direct member of a **Land**; nevertheless it is used in **veh.getweight()**. This member function is an implicit part of the class, inherited from its 'parent' vehicle.

Second, although the derived class **Land** now contains the functionality of **Vehicle**, the **private** fields of **Vehicle** remain private in the sense that they can only be accessed by member functions of **Vehicle** itself. This means that the member functions of **Land must** use the interface functions (**getweight()**, **setweight()**) to address the **weight** field; just as any other code outside the **Vehicle** class. This restriction is necessary so that the aspect 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.

[10]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.

In this example we assume that the class `Auto`, which represents automobiles, should be able to represent 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 its turn is derived from `Vehicle`. We speak here of **nested** derivation: `Land` is `Auto`'s direct base class, while `Vehicle` is 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 address allocated memory, these tools are needed.

6.2 The constructor of a derived class

As mentioned previously, a derived class inherits the functionality of 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 3.5).

The result in the above implementation is therefore that (a) the default constructor of a `Vehicle` is called, which probably initializes the weight of the vehicle, and that (b) subsequently the weight is redefined by calling `setweight()`.

The better solution is of course to directly call the constructor of `Vehicle` which expects an `int` argument. The syntax to achieve this, is to place the constructor to be called (supplied with an argument) following the argument list of the constructor of the derived class:

```
Land::Land (int wt, int sp)
    : Vehicle (wt)
{
    setspeed (sp);
}
```

6.3 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 part and the trailer have their own weight; but the `getweight()` function should return the combined weights.

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

- 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 `getweight()`:

```
int Truck::getweight () const
{
    return
        (                           // sum of:
            Auto::getweight () +    //   engine part plus
            trailer_weight          //   the trailer
        );
}
```

Note that in this function the call `Auto::getweight()` explicitly selects 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.

6.4 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 of 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 'cuz 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);
        char const *getserial () const;
        void setpower (int power);
        int getpower () const;
        void setfueltype (char const *type);
        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 is illustrated in the below listing. By using multiple derivation, the functionality of a `Auto` **and** of an `Engine` are swept 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 **derivation**, and **not** by the statement in the constructor of the class `MotorCar`. This means 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 **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, consider the unnecessary code duplication 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 better avoided by using derivation. Alternatively, when insisting on the **has** relationship and hence on composition, the interface functions could be avoided using `inline` functions.

6.5 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 which shall be discussed in this section.

6.5.1 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 poses however problems: what data should a statement like

```
a = v;
```

substitute for the fields `speed` and `name`, which are missing in the right-hand side `Vehicle`? Such an assignment is therefore not accepted by the compiler.

The following general rule applies: when assigning related objects, an assignment where some data are dropped is legal. An assignment where data would have to be left blank is however 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 can of course be explicitly defined, if needed. E.g., to achieve the correct working of a statement

```
a = v;
```

the class `Auto` would need an assignment function accepting a `Vehicle` as its argument. It would then 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
        .
    }
```

6.5.2 Conversions in pointer assignments

We define the following objects and one pointer variable:

```
Land
    l (1200, 130);
Auto
    a (500, 75, "Daf");
Truck
    t (2600, 120, "Mercedes", 6000);
Vehicle
    *vp;
```

Subsequently we can assign **vp** to the addresses of the three objects of the derived classes:

```
vp = &l;
vp = &a;
vp = &t;
```

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; this is the only functionality of a `Vehicle` and thereby the only functionality which can be accessed by using a pointer to a `Vehicle`.

The restriction in functionality has furthermore an important effect for the class **Truck**. After the statement **vp = &t**, **vp** points to a **Truck**; nevertheless, **vp->getweight()** will return 2600; and not 8600 (i.e., the combined weight of the cabin and of the trailer, 2600+6000) which **t.getweight()** would return.

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 function is available and executed. In other words, **C++** always implicitly converts the object which is pointed to to the type of the pointer.

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 unexpected behavior of the program may be the result.

6.6 Storing base class pointers

The fact that pointers to a base class can be used to address 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 then **contained** in the container class.

As an example we present here the class **VStorage**, which is used to store pointers to **Vehicles**. The actual pointers may be addresses of **Vehicles** themselves, but also may refer to derived types such as **Autos**.

The definition of the class is the following:

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

        // overloaded assignment
        VStorage const &operator= (VStorage const &other);

        // interface:
        // add Vehicle* to storage
        void add (Vehicle *vp);
        // retrieve first Vehicle*
        Vehicle *getfirst (void) const;
        // retrieve next Vehicle*
        Vehicle *getnext (void) const;

    private:
        // data
        Vehicle **storage;
```

```
        int nstored, current;
};
```

Concerning this class definition we remark the following:

- 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.

  The class could therefore be used as is illustrated in the following example:

  ```
  Land
      l (200, 20);            // weight 200, speed 20
  Auto
      a (1200, 130, "Ford");  // weight 1200 , speed 130,
                              // make Ford
  VStorage
      garage;                 // the storage

  garage.add (&l);            // add to storage
  garage.add (&a);

  Vehicle
      *anyp;
  int
      total_wt = 0;

  for (anyp = garage.getfirst (); anyp; anyp = garage.getnext())
      total_wt += anyp->getweight ();

  printf ("Total weight: %d\n", total_wt);
  ```

  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 type, is the greatest common denominator and hence can be used to compute the combined weight.

- The class `VStorage` furthermore 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 ();
printf ("%d\n", any->getweight ());
```

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()`.

There is, of course, also a remedy to this slight disadvantage. This will be discussed in the next chapter.

7 Polymorphism, late binding and virtuality

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 actual function will be called. This means that the code example as from section 6.6 which uses the storage class `VStorage`, will incorrectly compute the combined weight when a `Truck` object (see section 6.3) is in the storage — only one weight field, of the cabin 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`.

The opposite is however also possible. I.e., **C++** makes it possible that a `Vehicle *vp` calls a function `Truck::getweight()` when the pointer actually points to a `Truck`. The terminology for this feature of **C++** is **polymorphism**: it is as though the pointer `vp` assumes several forms when pointing to several objects. In other words, `vp` might behave like a `Truck*` when pointing to a `Truck`, or like an `Auto*` when pointing to an `Auto` etc..[11]

[11] In one of the StarTrek movies, Cap. Kirk was in trouble, as usual. He met an extremely beautiful lady who

A second term for this feature is **late binding**. This name refers to the fact that the decision **which** function to call (one of the base class or one of the derived classes) cannot be made at compile-time. The right function is selected at run-time.

7.1 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 6.1 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:
        // constructors
        Vehicle ();
        Vehicle (int wt);

        // interface.. now virtuals!
        virtual int getweight () const;
        virtual void setweight (int wt);

    private:
        // data
        int weight;
}

// Vehicle's own getweight() function:
int Vehicle::getweight () const
{
    return (weight);
}

class Land: public Vehicle
{
    .
```

however thereupon changed into a hideous troll. Kirk was quite surprised, but the lady told him: "Didn't you know I am a polymorph?"

```
    .
}

class Auto: public Land
{
    .
    .
}

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;
};

// 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 below the `(1)` mark, `Vehicle`'s function `getweight()` is called. In contrast, the statements under `(2)` use `Truck`'s function `getweight()`.

Statement `(3)` however will still lead to 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).

7.1.1 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.

7.1.2 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 right function can only be selected at 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.

The most common implementation is the following. An object which contains virtual functions holds as its first data member a hidden field, pointing to an array of pointers which hold 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 the 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.

A statement like `vp->getweight()` therefore 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 organization of the objects concerning virtual functions is further illustrated in the figure 6.

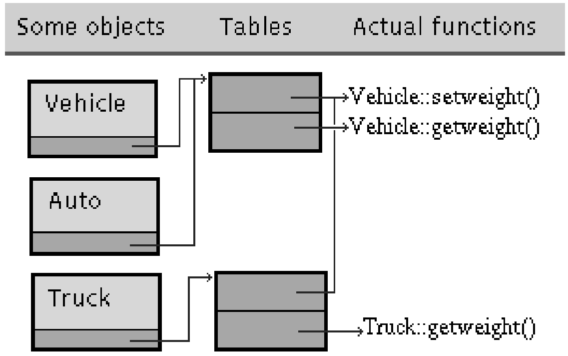

Figure 6: Internal organization objects when virtual functions are defined.

As can be seen from table 6, 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.

7.2 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 section 4.1), 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;

    // first try: if names unequal, we're done
    if ( (cmp = strcmp (name, other.name)) )
        return (cmp);
    // second try: compare by addresses
    return (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.

7.3 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;

    // next: if names unequal, we're done
    if ( (cmp = strcmp (name, other.name)) )
        return (cmp);
    // last try: compare by addresses
    return (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
}
```

7.4 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", "633688");

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 ()
{
}
```

7.5 Virtuality in multiple inheritance

As was previously mentioned in chapter 6 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

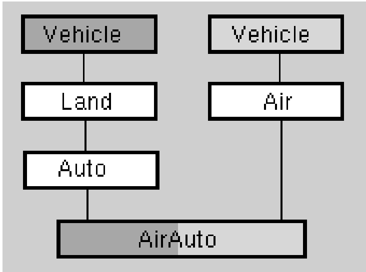

Figure 7: Duplication of a base class in multiple derivation.

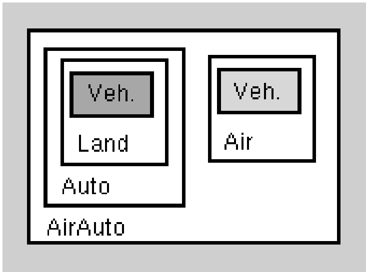

Figure 8: Internal organization of an `AirAuto` object.

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, section 6.1). Such a class would be needed to represent, e.g., a flying car[12]. An `AirAuto` would ultimately contain two `Vehicles`, and hence two `weight` fields, two `setweight()` functions and two `getweight()` functions.

7.5.1 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 7.

The internal organization of an `AirAuto` is shown in figure 8.

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

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

```
AirAuto
    cool;

printf ("%d\n", cool.getweight());
```

The question of which member function `getweight()` should be called, cannot be resolved by the compiler. The programmer has two possibilities to resolve the ambiguity explicitly:

- First, the function call where the ambiguity occurs can be modified. This is done with the scope resolution operator:

  ```
  // let's hope that the weight is kept in the Auto
  // part of the object..
  printf ("%d\n", cool.Auto::getweight ());
  ```

  Note the place of the scope operator and the class name: before the name of the member function itself.

- Second, a dedicated function `getweight()` could be created for the class `AirAuto`:

  ```
  int AirAuto::getweight () const
  {
      return (Auto::getweight ());
  }
  ```

The second possibility from the two above is preferable, since it relieves the programmer who uses the class `AirAuto` of special precautions.

However, besides these explicit solutions, there is a more elegant one. This will be discussed in the next section.

7.5.2 Virtual base classes

As is illustrated in figure 8, 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** 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:

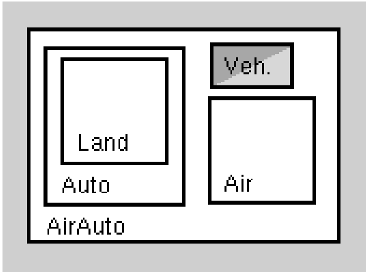

Figure 9: Internal organization of an `AirAuto` object when the base classes are virtual.

```
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 9.

Concerning virtual derivation we make the following final remarks:

- Virtual derivation is, in contrast to virtual functions, a pure compile-time issue: whether a derivation is virtual or not defines how the compiler builds a class definition from other classes.
- In the above example it would suffice to define either `Land` or `Air` with virtual derivation. That also would have the effect that one definition of a `Vehicle` in an `AirAuto` would be dropped. Defining both `Land` and `Air` as virtually derived is however by no means erroneous.
- The fact that the `Vehicle` in an `AirAuto` is no longer 'embedded' in `Auto` or `Air` has a consequence for the chain of construction. The constructor of an `AirAuto` will directly call the constructor of a `Vehicle`; this constructor will not be called from the constructors of `Auto` or `Air`.

Summarizing, virtual derivation has the consequence that ambiguity in the calling of member functions of a base class is avoided. Furthermore, duplication of data members is avoided.

7.5.3 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 6.3:

```
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
```

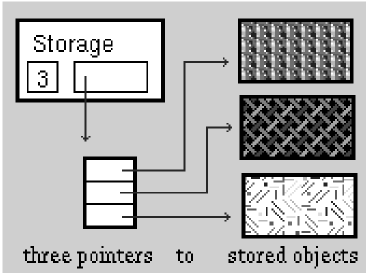

Figure 10: Internal organization of the class `Storage`.

```
        );
}
```

8 Concrete examples of C++

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

8.1 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.

8.1.1 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 10.

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 10) 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.

8.1.2 The class Storable

Using the above discussed approach we can now define the class `Storable`. The following questions are of importance:

- Does the class `Storable` need a default constructor, or possibly other constructors such as a copy constructor?

  The answer is no. `Storable` will be a bare prototype, from which other classes will be derived.

- Does the class `Storable` need a destructor? Should this destructor be (pure) `virtual`?

  Yes. The destructor will be called when, e.g., a `Storage` object ceases to exist. It is quite possible that classes which will be derived from `Storable` will have their own destructors: we should therefore define a `virtual` destructor, to ensure that when an object pointed to by a `Storable*` is deleted, the actual destructor of the derived class is called.

The destructor however should not be **pure** virtual. It is quite possible that the classes which will be derived from `Storable` will not need a destructor; in that case, an empty destructor function should be supplied.

The class definition and its functions are given below:

```
class Storable
{
    public:
        virtual ~Storable (void);
        virtual Storable *duplicate (void) 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 4.1. 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(*this)
    return (new (*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 ));
}
```

8.1.3 The class Storage

We can now implement the class `Storage`. The class definition is given below:

```
class Storage: public Storable
{
    public:
        // destructors, constructor
        ~Storage (void);
        Storage (void);
        Storage (Storage const &other);

        // overloaded assignment
        Storage const &operator= (Storage const &other);

        // functionality to duplicate storages
```

```
            Storable *duplicate (void) const;

            // interface
            void add (Storable *newobj);
            int nstored (void) const;
            Storable *get (int index);

        private:
            // copy/destroy primitives
            void destroy (void);
            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 8.1.1).
- 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 `Persons`.
- Internally, `Storage` defines two `private` functions `copy()` and `destroy()`. The purpose of these primitive functions is discussed in section 4.4.1.

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:

```
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]);
}

int Storage::nstored () const
{
    return (n);
}
```

8.2 A binary tree

This section shows an implementation of a binary tree in **C++**. Analogously to the classes `Storage` and `Storable` (see section 8.1) 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`.

8.2.1 The Node class

The class `Node` is an abstract (pure virtual) class, which defines the protocol for the usage of derived classes with a `Tree`. Concerning this protocol we remark the following:

- When data are stored in a binary tree, the place of the data is determined by some **order**: it is necessary to determine how the objects should be sorted. This requires a comparison between objects. This comparison must inform the caller (i.e., the function which places objects in a tree) whether one object is 'smaller' or 'greater' than another object.

  This comparison must lie with `Node`s: a `Tree` itself cannot know how objects should be compared. Part of the procotol which is required by `Node` is therefore:

  ```
  virtual int compare (Node const *other) const = 0;
  ```

  The comparing function will have to be implemented in each derived class.

- Similar to the storage of objects in the class `Storage` (see section 8.1), 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 (void) 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 (void) = 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 (void);
  ```

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

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

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

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

Node::~Node ()
{
}
```

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

8.2.2 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 (void);
        Tree (void);
        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);
        void inorder_walk (void);
        void postorder_walk (void);

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

        // 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 `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`.

8.2.3 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 also 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`.

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

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

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

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

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

```
}

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

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

Strnode::Strnode (char const *s)
{
    str = strdup (s);
    times = 1;
}

Strnode const &Strnode::operator= (Strnode const &other)
{
    if (this != &other)
    {
        delete str;
        str = strdup (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);
}
```

9 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.

9.1 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 no code **yet**; it only becomes 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 4.1 as illustration):

```
int main ()
{
    int
        a = 3,
        b = 16;
    double
        d = 3.14,
        e = 2.17;
    Person
        k ("Karel", "Rietveldlaan 37", "426044"),
        f ("Frank", "Oostumerweg 17",  "2223");

    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 1.4.10) 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; it must be present in, e.g., a header file.

9.2 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 8.

9.2.1 A template class: Array

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

```
#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 const &operator= (Array<T> const &other);

        // interface
        int size (void) 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 (void) 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 following:

- 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 4.1).

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 code of the template class is appended to the class definition, say in a header file `array.h`.

9.2.2 Using the Array class

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

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

#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
    .                               // isn't shown

    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.

9.2.3 Evaluation of template classes

In this chapter and in chapter 8 we have seen two approaches to the construction of container classes.

- The `Storable/Storage` approach from chapter 8 (see section 8.1) 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 from 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`, or `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 8 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.

We can therefore 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.