Learning C++

Aaron Isotton <aaron@isotton.com>

February 24, 2003

Contents

Foreword

It is a widespread belief that C and C++ are not good languages to learn programming in, but that one should start with "easier" languages first and then proceed towards the more "difficult" ones. Although I believe that this may be true for C, I do not think so for C++; thus this tutorial is targeted at people who have never programmed before.

If you find any mistakes in this tutorial, or think that some parts of it are not clear or precise enough, please contact me at `aaron@isotton.com`; if you don't understand something, it isn't *your* fault, it is *mine*.

Part I

Getting Started in C++

Chapter 1

Introduction

1.1 How a Program Works

A computer program is a sequence of commands executed by the Central Processing Unit (CPU) one after another. Those commands are generally very simple (like sums, multiplications, reading data from the Random Access Memory (RAM)), but are combined to do more complicated tasks. Typically a program consists of thousands to millions of such simple commands.

1.2 How Programming Works

1.2.1 Machine Code and Assembler

The most basic level of programming is to write *machine code* (single commands to the CPU) directly. This is very difficult as a modern CPU typically has hundreds of different commands, each of which does a different thing and has different interesting properties. But the main problem with this approach is not even the difficulty, but the fact that every different CPU model has a different command set, different strengths and weaknesses, and that it takes a *deep* knowledge of how the different hardware components and the used Operating System (OS) work.

To the other side, machine code is the only thing a CPU understands. Thus it is necessary to transform *every* program into machine code before supplying it to it. The question is only whether we want to do that by hand or use a program to do it.

Writing machine code means writing in binary[1] directly, *not* text. This is of course very tedious partly due to the fact that keyboards are made to write text with, and mainly that humans are more used to think in text than in binary. While this was popular once (as it was the only way to go) it is now almost forgotten.

[1] In programming, "binary" has two meanings. One of them is the traditional one, which means using the binary system (0 and 1). The other one is derived of the first, and is used in opposition to "text". Any non-text file (like, for example, a JPEG or an MP3) is called binary, and thus editing such a file is binary editing. In this case "binary" is used in the latter sense.

Assembler is nothing but textified machine code. Instead of having to remember the number of each command, you use names for the different commands. Of course each CPU still has different commands, and you need to have all the same knowledge as in writing machine code directly. Assembler is still widely used to write high-speed code, or for the kernel of an operating system (where the hardware is accessed directly), or for embedded systems. As assembler is plain text it cannot be executed by the CPU; it needs to be transformed into machine code first. This is done using a program called "Assembler" as well. To give you an idea, assembler looks more or less like this:

```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
pushl %ebx
movl $2,-4(%ebp)
movl -4(%ebp),%eax
leal 1(%eax),%edx
```

Generally assemblers offer a few other options as well to make programming a bit easier, but it is still very difficult. It is *definitely* a very bad idea to try (and fail) to learn programming using assembler.

Both of these approaches require a lot of skills and have the big disadvantage that it is necessary to virtually completely rewrite a program to port it to another CPU architecture or even OS.

1.2.2 High-Level Languages

Then there are the high-level languages. There are literally hundreds if not thousands of them, and it is impossible to give a description of all of them. Most of them share a similar approach, though. They are special languages similar to English (or to other spoken languages, although English is clearly predominant) which can be transformed into machine code doing what you asked for in that special language. For example, in BASIC the command

```
PRINT "Hello"
```

will print `Hello` on the screen. This is of course much easier to remember than a sequence consisting of things like the assembler code shown in 1.2.1.

There are a few things to remark about programming languages (and computers in general), though. A computer looks at everything in the form of ones and zeroes, *not* in form of "more or less". Thus it will do *exactly* what you tell it, not what you meant. And it will not accept anything with mistakes in it. For example, it will *not* execute

```
PRINT] "Hello"
```

even if it is obvious that the `]` was only a typo. Computers are not forgiving at all. If a command like `PRINT]` exists, it will execute that one even if it is obvious that it is not what you wanted.

But high-level programming languages are not only much easier to use than assembler or machine code; they have other advantages as well. As they do not rely on a particular CPU architecture, but use a more general approach (like

using the sign "+" instead of the "add" instruction of a particular CPU), they are much more *portable*. That means that the same program can be used on different platforms with no or little changes.

This still leaves us with one problem. How is a program written in some programming language (called the "source code") translated into machine code for a certain platform? There are two main solutions, both with their own advantages and disadvantages.

Interpreters

An *interpreter* is a program which reads source code, and executes the commands in it while it is reading it. There is no real translation in traditional interpreters; basically the interpreter knows what to do when it finds a certain command. For example, when a BASIC interpreter finds a `PRINT` command it knows that it must print the text which follows it on the screen, and so on.

This approach has the following main advantages:

- A program can be run immediately after being written.
- Everybody with a suitable interpreter can run your program, independently from your platform. It is not necessary to ship different versions of your program for different platforms [2].
- Everybody can read, analyze and modify your source.

But it has several disadvantages as well, and that's why interpreted languages are not so widely used anymore today:

- Interpreted programs are *much* slower than compiled ones. 100 times slower is no rarity. Under some circumstances this might not be acceptable. Although this sounds like a horrible flaw, it isn't (anymore) because nowadays' computers are often thousands of times faster than what you'd need.
- Everybody can read, analyze and modify your source.
- Everybody who wants to use your program must have a suitable interpreter.
- Some errors in your program (like `PRINT]` instead of `PRINT`) may be detected only after a program is shipped, because they are only found if the program ever reaches that point.

Interpreted languages are nowadays used mainly where the same program needs to run on a wide variety of platforms. Examples include BASIC, JavaScript and some others. JavaScript, for example, must be executed on totally different platforms.

Another application of interpreted languages is scripting, used in many advanced applications. (If you don't know what scripting is, don't worry because you don't need to.)

[2] This is not true with modern interpreted languages, but I'm not nitpicking here.

Compilers

The other approach are compilers. A compiler takes the source code and translates it into machine code, generally generating some kind of "executable file" (the ".exe" files under windows, or some of the files with the "executable" flag under Unix). The program can be executed only after the compilation has finished. Of course, if the compilation fails because there are errors in the source of for some other reason, no executable is generated and it can thus not be executed either.

The main advantages of compiled languages are the following:

- A compiled program is real machine code and runs directly on the CPU. It is thus generally very fast.
- It is not possible to read and analyze the source code if you only have the executable. Thus, people can not steal your code so easily.
- A compiled program does not need any supporting programs like interpreters [3]. It is a stand-alone program.
- Compiled languages are generally more powerful than interpreted ones.
- Many errors which might not be noticed in an interpreted program will be detected when you try to compile the source.

But, as for interpreted languages, there are also some drawbacks:

- It is not possible to read and analyze the source code if you only have the executable. Thus, people cannot fix errors or improve your program easily.
- You need to compile a different executable for most platforms. This might involve substantial changes to the source.

Compiled languages are used for most software applications, like office suites or HTML editors or anything else you can think of. They are the only way to go for high-speed applications like 3D games or photo retouching programs. The most popular compiled languages are C and C++.

Borderline Cases

There are also some languages which are somewhere in between the compiled and interpreted ones. The most famous of them is Java, which is compiled into some kind of virtual machine code, which is then executed on a Java Virtual Machine (which is a special software). This way Java achieves a considerable speed and a high portability at the same time. There are other drawbacks, though.

Another noteworthy language is Perl. It is an interpreted language, but there are also compilers available for it. Moreover, the Perl interpreter has some characteristics of a compiler and can thus not be put into either of the two categories.

[3] There are exceptions, like Microsoft Visual Basic, which is a compiled language but needs quite substantial runtime libraries to run. In fact, most programmers do not consider Visual Basic a programming language because, they think that it is just a toy, which is true.

1.3 Issues of Programming

From what I've told you until now, programming does not seem like a very difficult task: you just need to know a programming language and can just start writing things in it.

This is true, but there is more to it anyway. First of all, learning a programming language is not exactly as easy as you might think; then, there are other issues as well, like portability and speed.

The most interesting of those is portability. A program is "portable" if it can be easily changed to run on another platform. This is generally possible only if you do not rely on any platform's special features or architecture. For example, if you use a feature which only Unix has, your program is not portable because it won't run on a Mac, and vice versa.

1.4 C++

C++ is a compiled high level language, with features for both object-oriented and imperative programming. It comprehends most of the functionality of C (which is another programming language). It is very widely used and very powerful, and there are lots of good compilers available on almost any platform.

Programs written in C++ are typically comparably fast to equivalent programs written in C, but simpler. To give you an idea of what "fast" means, the Linux Kernel, most of the programs by the Free Software Foundation and all games by id software (as Doom I, II and III, Quake I, II and III as well as many others) are written in C.

1.4.1 The Standard

One very important aspects of C++ distinguishing it from most other programming languages is the fact that it is *standardized*. This means that there is an internationally accepted standard (currently ISO/IEC 14882-1998) for C++ specifying what its features are and how it works. As all modern compilers are built trying to follow the standard as closely as possible, if you write a program in Standard C++ (without any compiler or platform specific extensions) it is very likely that you will be able to compile it with no or little changes under another platform and compiler. This makes portability much much easier.

1.5 What You Need

Before you start programming, there is some software you need. Fortunately you can get it all for free, but if you want you can also spend lots of money on it.

1.5.1 A Compiler

Of course you need a C++ compiler for your platform. There are many of them, but I'd recommend you use a recent version of gcc if you are working under Unix or a port of it for any other platform. (The windows port of gcc is called mingw).

You also need to know how to invoke your compiler; please refer to your compiler's manual for that.

1.5.2 An Editor

Then, you need a good text editor to edit the source code. I'd recommend emacs and vim for Unix, or just any other editor you like if you are using another platform.

Chapter 2

Hello World!

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

int main()
{
    cout << "Hello World!\n";
}
```

This is the classic first program you write in a programming language, but especially in C or C++. It is called "Hello World" because it prints the text `Hello World!` on the screen and exits.

Please notice that C++ is a *case-sensitive* language. In other words, `main` and `Main` are *not* the same things. Do not forget that when you want to try something out.

But now let's analyze it line by line. The first line is

```
#include <iostream>
```

and it is a *preprocessor directive*. (All lines starting by `#` are preprocessor directives). It is not a "real" C++ command, just a command to the compiler telling it to take the file `iostream` and to insert it instead of the directive. This is necessary because there the file `iostream` introduces some new commands we need later.

The second line

```
using namespace std;
```

tells the compiler that we will be using things which are in the namespace `std`. A *namespace* is a bit like a directory with files in. We tell the compiler that we will be using these files without every time specifying the namespace. This is necessary because the things in the file `iostream` are in the namespace std. Don't worry if you do not understand this now.

The third line

```
int main()
```

declares the main function (it is called a "declarator"). This is necessary in every C++ program, as execution starts in the main function. I'll explain that

better later on. For the time being just remember that every C++ program has a main function.

The fourth line

```
{
```

marks the beginning of the *body* of the function main.

The fifth line

```
cout << "Hello World!\n";
```

is the only statement inside the function body. It prints `Hello World!` on the screen. To do that, it uses `cout`, which in C++ means standard output (generally the screen) and sends it the string it wants to print. We had to include `iostream` before exactly because of this: `cout` is one of the things defined in that file. If we hadn't used `using namespace std;` we'd have to write `std::cout` instead.

You might wonder what the `\n` at the end of the string means; it is a *newline character*, a character which makes a new line start. This way the output under Unix might look something like this:

```
$ ./helloworld
Hello World!
$
```

while without the `\n` it would be something like this:

```
$ ./helloworld
Hello World!$
```

Of course, the same thing is true for Windows as well, with the difference that your prompt will be `C:\>` or something like that.

Finally, the line

```
}
```

marks the end of the body of the main function.

Do not worry if you do not fully understand everything right now; this was just a quick overview of how a simple program works. We'll dive into the details later.

2.1 The Minimal C++ Program

The minimal (shortest possible) C++ program is the following:

```
int main(){}
```

As you can see, it only contains the definition of the main function with an empty body. In fact that is enough for a perfectly valid C++ program; it does not need to include any other files because it does not need any functionality.

Of course, the minimal program does nothing.

Chapter 3

Doing Things With Objects

In C++ most data is stored in *objects*. An object is something with a type and a status, and, optionally, a name.

3.1 What is an Object?

3.1.1 Type

The type of an object is its kind. For example, the object "Gandalf" might be of type "Wizard"[1]. Or the object "Pi" might be of type "Number", and so on.

3.1.2 Status

The status of an object depends on its type. The status of the above mentioned object "Pi" might be its value (3.1415926...). The status of an object of type "Wizard" might be a lot more complicated; it might include the wizard's name, his current position, what he is doing, his mood, whether he is dead or alive, and so on. You will often hear an object's status referred to as "value" if the object is simple, like for example a number.

3.1.3 Name

An object *might* have a name, but it will not necessarily have one. For example, an object of type "Wizard" might have the name "Gandalf". But there is no relationship at all between an object's status and its name; it would be very well possible having an object of the type "Wizard" containing data about Gandalf but with the name Sauron (or Frodo, or Bilbo, or whatever you prefer)[2].

3.2 Objects in C++

In C++ there are several different types of objects, and it is possible to create as many new ones as you like. For example, there are lots of different types

[1] For those of you who haven't read [LOTR], Gandalf is a powerful Wizard.

[2] Again, who hasn't read [LOTR] should know that Sauron is another wizard (an enemy of Gandalf), while Bilbo and Frodo are hobbits (which are quite different from wizards).

dealing with numbers (because computers are designed to work with numbers), but there isn't even one to deal with wizards. To overcome that horrible lack, we'll design one further on.

To *create* an object you must first *declare* it, specifying its type, its name, and optionally its initial value. This is done as follows:

```
int number_of_wizards;
```

where `int` and `number_of_oranges` are the type and the name for the new object, respectively. Of course, you could use any other type instead of `int` and any other name instead of `number_of_wizards` as long as they are both valid.

If you want to give the newly created element a new value while you create it, you can use the following syntax:

```
int number_of_wizards = 1;
```

Giving an object an initial value during its declaration is called *initializing*.

3.2.1 Valid Names

Not all names you can think of are valid in C++. A valid name may be composed only by alphanumeric character (`a`–`z`, `A`–`Z` and `0`–`9`), or the underline character "`_`". The first character may not be a numeric character, though. Thus, the following names are valid:

```
a
a1
s0
hello_world
GandalfRulez
```

while the following ones are not:

```
a b c
9oclock
number-of-wizards
```

The first one of them is invalid because it contains whitespace characters; the second one starts by a number, and the third one contains hyphens which are not allowed either.

There are some names which are allowed but which you shouldn't use either. All names containing a sequence of two underline characters at any position (`__`) are reserved for the implementation, and so are any names starting by an underline character followed by an upper-case letter. You *can* use them, but you should not. Avoid problems and don't.

3.2.2 Fundamental Types

The most important[3] types in C++ are `int`, `double` and `std::string`.

`int` can hold both positive and negative integers (like 1, 2, 3...). It is the type you should generally use for integer values.

[3] In my opinion.

`double` can hold floating-point (decimal) numbers (like 1.32 or -.3). Because of some problems with binary rounding[4] `double` should be used *only* if you have to use fractional values. If anyhow possible, use `int`!

`std::string` is fundamentally different from `int` and `double`, not only because it holds strings (text) and not numbers, but because it is not a builtin type. "Not a builtin type" means that it is not a type which belongs to the language, but, like for `cout`, a special file must be included if you want to use it. That file is `string`, and, as for `cout`, you must either use `using namespace std;` or put a `std::` in front of `string` before using it. I strongly recommend the first option for the time being.

3.3 Assigning

Now that you know everything about the declaration of objects[5], it is about time to start doing things with (and to) them.

The first thing to do is assignment. It works like this:

```
number_of_wizards = 4;
```

Basically it's the same as initializing, but with an object which already exists (while in initialization you give it a value while it is created). Of course, the two objects must be of the same type, or the compiler must know how to convert between them. (More on that later.) Have a look at the following full example:

```
int main() {
    int a = 12; // a is now 12 (initialization)
    a = 32;     // a is now 32 (assignment)
    a = a + 1;  // and now 33 (assignment)
    a = a / 2;  // and now 16 (assignment)
}
```

First `a` is declared and initialized with the value of 12. Then it is *assigned* the value 32. After that it is assigned its own value plus one, thus 33. And then it is assigned its own value divided by two, thus... 16? But why not 16.5? The answer is simple. `a` is an `int`. `int`s hold integer values only. If an `int` is assigned a fractional value, it is rounded towards 0.

3.4 Input and Output

It is of course nice to have a program calculate things for you; it would be a lot nicer, though, if the values could be entered when the program is run and not hard-coded into the program. Doing that is quite easy; in fact, we've already done it (in part) in the "Hello World" program, which could print text on the screen.

To send data to standard output (generally the screen) in C++ you use `cout`, as follows:

[4] This is a really hairy subject so I won't go into it here.

[5] That was sarcastic. Your knowledge is still closer to zero than to infinity.

```
// This program demonstrates how to use cout
#include <iostream>
using namespace std;

int main() {
    cout << "Hello ";
    cout << "World.\n";
    cout << "Hello World.\n";
    int a = 5;
    cout << "a = ";
    cout << a;
    cout << "\n";
    int b = 23;
    cout << "b = " << b << "\n";
}
```

As you can see, you can use `cout` to print text, `int`s, and almost anything else you want. Notice how you can send more than one object to output at the same time using multiple `<<` signs, and that `cout` *does not* begin a new line by default. You must use `\n` for that. As `cout` is defined in `iostream` we have to include it.

The `//` in the first line is a *comment*. Comments are ignored by the compiler and used to write reminders to yourself into a program. C++ has two different types of comments. The ones used in this example are the "C++-style" comments, begin by `//` and end at the end of the line. The other ones are "C-style" comments and begin by `/*` and end by `*/`. They can thus span multiple lines.

Input works just like output: use `cin` instead of `cout` and `>>` instead of `<<` and you're done. Or almost. Look at this simple example:

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

int main() {
    int year, age;
    cout << "What year were you born? ";
    cin >> year;
    age = 2002 - year;
    cout << "You are " << age << " years old.\n";
}
```

This program takes the year of your birth and calculates your age. You might need to update the program if it is outdated in the moment you read it. Notice that also `cin` is defined in `iostream`.

There are a few peculiarities of `cin` though. Look at the following program:

```
// This program shows how cin only reads until the
// first whitespace character.

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

```
int main() {
    cout << "Please enter your name: ";
    string name;
    cin >> name;
    cout << "Your name is " << name << "\n";
}
```

First of all, notice how we used the `string` type to read in text strings. We need to include `string` to do so. If you enter `Joe` when it asks you for your name it will nicely display `Joe` as your name, just as it should. But if you enter `Joe Programmer` it will display `Joe` again, just dropping the `Programmer` part! Why that?

The answer is simple: `cin` only reads until the first whitespace character, then it stops. This might seem absurd but it isn't as you will see later. This is fine if you want to read in a number, because it will not contain any whitespace; it might be a problem with strings though. To read in a full line, you use `getline` like this:

```
// This program shows how getline works.

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

int main() {
    cout << "Please enter your name: ";
    string name;
    getline(cin, name);
    cout << "Your name is " << name << "\n";
}
```

Of course, `getline` can be used with `string` only, as it wouldn't make any sense with other data types.

Chapter 4

Make Decisions

Conditionals are special statements which execute a statement depending on a condition (like "if you pay, you get the beer"). There are two conditionals in C++: `if` and `switch`.

4.1 `if`

The `if` statement is used just as in English:

```
// How 'if' works

#include <iostream>
using namespace std;

int main() {
    cout << "Enter your age: ";
    int age;
    cin >> age;

    if (age < 20)
        cout << "You are still young!\n";
}
```

As you can see, if the condition in the parentheses is true the following statement is executed, otherwise it isn't. But there is more to `if`:

```
// More about if

#include <iostream>
using namespace std;

int main() {
    cout << "Enter your age: ";
    int age;
    cin >> age;

    if (age < 20)
```

```
        cout << "You are still young!\n";
    else
        cout << "You are not so young anymore.\n";
}
```

Here the `else` statement was used. If the condition of the `if` is true, the statement after the `if` is executed. Otherwise, the one after the `else` is executed.

4.1.1 Compound Statements

In C++, every *simple statement* can be replaced by a *compound statement* or *block*. But first of all: what is a simple statement? The short answer: everything followed by a semicolon except a declaration. This is not 100% true, but good enough as a rule of thumb. A compound statement (or block) is a group of simple statements, enclosed by braces (`{}`). In other words, instead of a single statement after an `if` block you can have many "grouped" statements which are executed together, like here:

```
if (age < 20) {
    // Put as many instructions as you want here
}
else {
    // Do other things...
}
```

4.2 switch

The other conditional is `switch`. It is not as widely used as `if` but very useful under some circumstances. It is used to analyze a variable, compare it to a series of constants and execute the associated code, as follows:

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

int main() {
    cout << "What would you like to do:\n";
    cout << "1. Add an entry to the address book\n";
    cout << "2. Look up an address\n";
    cout << "3. Remove an entry\n\n";
    cout << "Your choice: ";
    int selection;
    cin >> selection;

    switch (selection) {
    case 1:
        cout << "Sorry, this feature has yet to be "
             << "programmed.\n";
        break;

    case 2:
```

```
        cout << "Sorry, this feature was not yet "
             << "implemented.\n";
        break;

    case 3:
        cout << "Access denied.\n";
        break;

    default:
        cout << "You can't do that. You must choose "
             << "1, 2 or 3.\n";
        break;
    }
}
```

As you can see, one of the `case` statements is executed if the constant matches the variable passed to `switch`, and the `default` statement is executed if it no one matches. The `default` statement is optional; if there is none, and the variable doesn't match any of the constants, no one of the statements is executed.

There are some drawbacks with the `switch` statement, though. The first one is that the execution "falls through" the `case` clauses. This means that when the end of a `case` clause is reached, the program will *not* jump to the end of the entire `switch` block and continue execution after it as one would expect, but just continue in the next `case` clause until it reaches the end of the `switch` block. To avoid that, you use the `break` statements, which tell the compiler to jump to the end immediately. That's why it is very important *never* to forget the `break` statements. It is completely legal and correct not to use them, and under some circumstances it might be even useful; but generally you need them.

The second drawback is that only constants can be used in the `case` clauses. For the simple programs we are writing at this stage this is not a problem; further on you will no doubt need to compare a variable to other variables, and not constants; to do so, just use `if` and the `==` operator, which I'm just going to talk about.

There are also quite a lot of other problems with `switch` statements, but I'll cover them later.

4.3 The Most Important Operators

To be able to use the `if` statement, you need some *operators*. One of them (`>`) was already used in the example program; but there are several others you need to know about:

1. `==` is the equality operator. `a == b` is true if `a` is equal to `b`, false if it isn't. Do not confuse this operator with the assignment operator `=`; `a = b` assigns the value of `b` to `a`.

2. `!=` is the opposite of `==`. `a != b` is true if `a` is not equal to `b`, false otherwise.

3. `>`, `<`, `>=`, `<=` are the greater, less, greater or equal to and less or equal to operators, respectively. They work just as you'd expect.

4. `&&` is the logical and operator. It is true if both expressions to its left and its right are true. For example, `a > 10 && a < 20` is true if `a` is greater than 10 *and* less than 20 (in other words, if it is somewhere between 11 and 19). Make sure you do not confuse this with operator `&`.

5. `||` is the logical or operator. It is true if at least one of the two expressions is true. For example, `a < 10 || a > 100` is true if `a` is less than 10 or greater than 100. Make sure you do not confuse this with operator `|`.

6. `!` is the logical not operator. It negates what follows. For example, `!(a > 10)` is the same thing as `a <= 10`.

4.3.1 Precedence and Parentheses

There is a problem with operators though. As long as there is only one operator, like in `a < 10` everything is clear. But what if you write `a < 10 || a > 10 && !a >= 10 - a`? Obviously it is not clear what that expression does, because it is not clear what is executed first. There "operator precedence" comes in. In C++ every operator has a certain precedence; operators with a higher precedence are evaluated first. As a general rule, the math operators (`+`, `-`, `*`, `/`) have a higher precedence than the comparison operators (`>`, `<`, `>=`, `<=`), and these have a higher precedence than the logical operators (`\&\&`, ——). You can use parentheses to influence the order in which operators are executed, like this:

```
if ((a > 10) || (a < -10)) {
    //...do whatever you want
}
```

This is the exact same thing as

```
if (a > 10 || a < -10) {
    //...do whatever you want
}
```

but it is better to use too many parentheses than too few if you are not sure.

4.4 Scope

With compound statements and the `if` statement it is necessary to introduce a new concept: *scope*.

Scope is yet another aspect of an object. It is the part of the program from where an object can be accessed. For example, in

```
int main() {
    int i;
}
```

the variable (or object, if you prefer) `i` has a scope which goes from the declaration to `}`. In other words, it can be accessed only between the declaration and the closing brace. That might not sound very sensible, because of course it would not make any sense to access it before it was declared, and you cannot put anything after the closing brace anyway. But look at this:

```
int main() {
    if (1 == 1) {
        int i;
        i = 3; // This is fine
    }

    i = 4; // This causes an error
}
```

The condition `if(1 == 1)` is of course always true because `1` is always equal to `1`. But that's besides the point.

As you can see, we declare `i` inside the `if` block. This is no problem. Then we assign `3` to it. This is no problem either. But we cannot assign `4` to it later *outside* the `if` block because the scope of `i` goes from the declaration to the closing brace of the `if` block.

Opposed to that the following will work just fine, because the scope of `i` goes from the declaration of `i` to the closing brace of the `main` block.

```
int main() {
    int i;
    if (1 == 1) {
        i = 3;
    }
    i = 4;
}
```

It is also possible to declare (and optionally initialize) object outside any block, like this:

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

int i = 4;

int main() {
    cout << "i = " << i << "\n";
    i = 232;
    cout << "i = " << i << "\n";
}
```

An object declared like this is said to have "global scope", or just called a "global object" because it is available everywhere[1].

As every object has a scope, it is of course possible to declare objects with the same name in different scopes, like this:

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

int main() {
    cout << "1. Calculate a sum\n";
```

[1] Don't take this too literally. You'll see later why.

```
    cout << "2. Calculate a product\n";
    cout << "Your choice: ";
    int selection;
    cin >> selection;

    if (selection == 1) {
        cout << "Enter the two numbers to sum "
             << "separated by a space: ";
        int first, second;
        // Notice how cin can be used to input more
        // than one value in a single call
        cin >> first >> second;
        cout << "The sum is: " << first + second
             << "\n";
    }
    else {
        cout << "Enter the two numbers to multiply "
             << "separated by a space: ";
        int first, second;
        cin >> first >> second;
        cout << "The product is: " << first * second
             << "\n";
    }
}
```

Notice that the two objects are *not the same thing*. They just happen to share the same name, but they are completely independent, just as two people with the same name have two independent bodies, brains, arms, and whatever else you might think of.

4.4.1 Lifetime

Every object has a *lifetime*. The lifetime of an object starts when it is declared and ends when it goes "out of scope". This means that an object is *created* or *constructed* when it is declared, and it is *destroyed* when it goes out of scope.

```
int main() {
    int i; // i is created here

    if (true) {
        int j; // j is created here
    } // j is destroyed here

} // i is destroyed here
```

If you have understood the concept of scope, this is quite logic. When an object goes out of scope it can not be used anymore, and is thus destroyed.

4.4.2 Hiding

Another concept strictly linked to scope is *hiding*. Hiding happens when an object with the same name as another object is declared inside the scope of the

first one, as follows:

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

int main() {
    // Declare the object i.
    int i = 5;

    // This will print 5, of course.
    cout << "i = " << i << "\n";

    if (true) {
        // Now we declare *another* object i. It is
        // *not* the same i as before, but a completely
        // independent one, and it will hide the first
        // i (the one with the value 5).
        int i = 7;

        // This will print 7 because the first i (with
        // the value 5) is hidden by the second i (with
        // the value 7).
        cout << "i = " << i << "\n";

        // The scope of the second i ends here.
    }

    // This is 5 again, because we are referring to the
    // first i again.
    cout << "i = " << i << "\n";
}
```

It is of course not a very good idea to do so, because it is confusing and it is not possible to access the hidden object inside the scope of the hiding object. It is, though, an important concept.

4.5 Examples

Here a few a bit a longer program for your pleasure.

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

int main() {
    // First display the menu.
    cout << "What do you want to do:\n";
    cout << "1. Calculate the area of a square\n";
    cout << "2. Calculate the area of a circle\n";
    cout << "Your choice: ";
```

```
    // Take the user's selection.
    int choice;
    cin >> choice;

    // Do the right thing based on the user's
    // selection.
    switch (choice) {
    case 1: {
        // We need braces here because otherwise we
        // cannot declare the variable 'side' below.
        cout << "Please enter the side length: ";
        double side;
        cin >> side;

        // We do not accept negative side lengths.
        if (side < 0)
            cout << "There can be no squares with "
                 << "negative side lengths. Bye.\n";
        else
            cout << "The area is " << side * side
                 << ".\n";
        break;
    }

    case 2: {
        // This is all the same as above, just for
        // circles.
        cout << "Please enter the radius: ";
        double radius;
        cin >> radius;
        if (radius < 0)
            cout << "There are no circles with "
                 << "negative radiuses. See you.\n";
        else
            cout << "The area is "
                 << radius * radius * 3.1415926
                 << ".\n";
        break;
    }

    default:
        // The user entered an invalid selection;
        // display an error message.
        cout << "Your selection isn't valid.\n";
        break;
    }
}
```

4.6 Exercises

Yes, exercises. You should do them. Really. You will not learn a programming language (or anything which has to do with computers) just reading. You must try it out, make mistakes, fiddle with it.

1. Write a program asking for an integer number and saying whether it is positive or not. (Zero is not positive).
2. Modify the previous program so that it says whether the number is positive, negative or zero.
3. Write a program displaying a menu like some of the presented programs to let the user select one among some (at least 4) recipes, and display the recipe the user has chosen.
4. Extend the previous program to detect if the user has made an invalid selection, and display an error message if he has done so.

Chapter 5

Loops

Computer programs are often used for repetitive tasks: generating sequences of numbers, serving thousands or millions of web pages, and so on.

Until now, if you wanted to print the same text three times you'd have to type the same statement three times:

```
cout << "Hello World!\n";
cout << "Hello World!\n";
cout << "Hello World!\n";
```

Of course, that is not a good idea. You can do it if you need to print it out three times. But what if you had to print all the numbers from 1 to 1,000,000? Or ask the user for a list of 32 names? Or what if you didn't know how many times something has to be done as you were programming it? That's where loops come in.

A loop is a statement which repeats another statement until a condition is met. That other statement can of course be a block, or even another loop, or whatever else you wish.

5.1 `while`

In C++ the simplest loop is the `while` loop. It is used as follows:

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

int main() {
    int i = 0;

    // Execute the loop until (i < 10) is not true
    // anymore, in other words until i >= 10
    while (i < 10) {
        // Print the number
        cout << i << " ";

        // Increase i by one
        i = i + 1;
```

```
    }
}
```

As you can see, `while` was used to repeat a block, not a single statement. That was necessary because we want it to repeat *all* the statements inside the block, not just a single one. It is, though, totally correct to use a single statement instead.

To give you an idea, a `while` loop works like this:

1. The condition is evaluated. If it is true, execution continues in 2, otherwise the loop is ended.
2. The statement is executed. Then execution jumps back to 1.

Thus, if the condition is false from the very beginning, the statement to repeat is *never* executed. For example, in

```
int i = 100;
while (i < 10)
    cout << i;
```

the `cout` is never executed, because `i` is greater than 10 from the very beginning.

5.2 do...while

The `do...while` loop is used as follows:

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

int main() {
    int i;

    // Keep asking until the number is greater than 10
    do {
        cout << "Please enter a number greater than "
             << "10:";

        cin >> i;
    } while (i > 10);

    cout << "You entered: " << i << "\n";
}
```

It is very similar to the previous one, with the difference that the condition is checked at the end of the loop and not at the beginning. It is executed as follows:

1. The statement is executed.
2. The condition is evaluated. If it is true, execution jumps back to 1; otherwise, the loop exits and execution continues normally.

The difference between the `while` loop and this loop is that in the first one the statement might not be executed at all (if the condition is false from the very beginning), while in the second it is always executed at least once.

5.3 for

The `for` loop is the most complicated, but also the most used loop in C++. It is thus very important that you understand perfectly how it works.

The `for` loop is used as follows:

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

int main() {
    for (int i = 0; i < 10; ++i) {
        cout << i << " ";
    }
}
```

As you can see, it is used like the `while` loop, but it takes three statements separated by semicolons.

The first statement is executed only once, before the loop starts. In this case it is used to declare and initialize the variable `i`, but you can do there whatever you like. The statement is not evaluated; this means that it does not matter whether it is true or not.

The second statement is the condition, which is checked (like in the while loop) before each iteration (each repetition of the statement of a loop is called an *iteration*); if it evaluates to false the loop is exited. In this case the condition is `i < 10`, thus the loop keeps iterating until `i` is greater or equal to 10.

The third statement is executed after each iteration. In this case it is `++i`. The operator `++` increases a variable by one (increasing something by one is called *incrementing*); it is, in this case, the same thing as `i = i + 1`. (Notice that this is true in this case, but not generally; you will learn more about the difference later.)

In other words, the previous `for` loop is a convenient way to do the same thing as here:

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

int main() {
    int i = 0;
    while (i < 10) {
        cout << i << " ";
        ++i;
    }
}
```

There is an important difference, though.

If you declare an object in the first statement in a `for` loop, that object is scoped to the three statements, plus the statement to repeat (the entire block if that is a block). Look at this:

```
for (int i = 0; i < 10; ++i) {
    // i can be accessed from here
    cout << i << " ";
```

```
} // Here i goes out of scope

// Thus, it can not be used anymore here
i = 10; // ERROR!
```

This is *very important*. It means that you can use several `for` loops one after another with a variable of the same name without having them clash. (Two variables *clash* if they share the same name and have the same scope. The compiler outputs an error message because it cannot know which variable you're referring to).

Although the `for` loop is mostly used in the form given above, it can be used for several other things as well. For example, you could modify the program simply using `i = i + 2` to display all even numbers, or `i = i * 2` for squares (you'd have to start at 1 and not at 0 to do so, though, because `0 * 2` will always be 0).

5.4 `break` and `continue`

There are two very useful statements to interrupt the normal execution of a loop: `break` and `continue`. We already know `break` from the `switch` statement; in fact it acts very similarly inside a loop. It immediately exits it and jumps to the first statement after the loop. It is generally used to prematurely exit a loop even if the loop's condition is still true.

```
// Let the user enter 100 positive numbers, or a
// negative number

#include <iostream>
using namespace std;

int main() {
    cout << "Enter 100 positive numbers, or a "
         <<" negative number to abort.\n";

    // Notice that here we declare i *outside* of the
    // loop. You'll see later why.
    int i;

    // Here we start counting at 1 and not at 0 because
    // otherwise the program would ask for number #0,
    // then for number #1, but we want it to start at
    // 1.
    for (i = 1; i <= 100; ++i) {
        cout << "Enter the number #" << i << ": ";
        int n;
        cin >> n;
        if (n < 0)
            break;
    }
```

```
    // If we hadn't declared i outside the loop, we
    // couldn't access it here because it'd be out of
    // scope.
    if (i == 100)
        cout << "You are a real man.\n";
    else
        cout << "You stopped after " << i
             << " numbers, coward!\n";
}
```

The other statement, `continue`, is slightly more complicated but just as useful. It skips all the rest of the statements inside the loop and goes immediately to the next iteration. It is often used to skip the execution if a certain condition is met. Have a look at the following program which prints the numbers from 0 to 100 except the multiples of 7:

```
// Print all numbers from 0 to 100 except the multiples
// of 7.

#include <iostream>
using namespace std;

int main() {
    for (int i = 0; i < 100; ++i) {
        // Skip all the multiples of 7
        // Operator % calculates the modulo (remainder)
        // of a division; if (i % 7) is equal to zero
        // this means that i is a multiple of 7.
        if ((i % 7) == 0)
            continue;

        cout << i << " ";
    }
}
```

As you can see, here `continue` is used to skip all the multiples of 7.

5.5 Examples

The following program demonstrates the use of *nested loops*:

```
// This program prints out a multiplication table as
// follows:
// 1  2  3
// 2  4  6
// 3  6  9
//
// You could use more numbers than just from one to
// three, but then there'd be problems with the
// alignment (because there'd be numbers with one and
// with two digits).
```

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

int main() {
    for (int y = 1; y <= 3; ++y) {
        for (int x = 1; x <= 3; ++x) {
            cout << x * y << " ";
        }
        cout << "\n";
    }
}
```

This one here is a bit more useful. It calculates factorials. (The factorial of a positive integer number n is mathematically written as $n!$ and is the product of all numbers from 1 to n; mathematically, $n! = 1 \times 2 \times \ldots \times n$. So $7! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 = 5040$.)

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

int main() {
    cout << "This program calculates the factorial "
         << "of n.\n";
    cout << "Please enter n: ";
    int n;
    cin >> n;

    if (n < 0)
        cout << "n must be non-negative.\n";
    else {
        int factorial = 1;

        for (int i = 1; i <= n; ++i)
            factorial = factorial * i;

        cout << n << "! = " << factorial << "\n";
    }
}
```

The only problem here is that `int`s cannot hold infinitely large values (in fact, they're *guaranteed* to hold only values from -32767 to 32768, although generally they can hold larger values); thus if you enter too large values there will be an *overflow*[1] and the output will most probably be garbage. In this specific case, the result is guaranteed to be correct only if $n \leq 7$, because $8! = 40320$ and is thus bigger than 32768. I'll return on this subject later.

[1] An *overflow* happens when you try to put a value into a variable which cannot hold such big values. In the case of `int`s it is not defined what will happen if an overflow happens. Most probably the result will be garbage. The opposite of an overflow is an *underflow*; this happens when you try to put a too small value into a variable (for example assigning -100000 to an `int` might cause underflow).

5.6 Exercises

1. Write a program which prints all numbers from 0 to 100.
2. Modify the previous program to display all even numbers from 0 to 100.
3. Now modify it to display all the odd numbers from 0 to 100.
4. Write a program asking the user for the upper and the lower bound and then displaying all the numbers between them. It should work somehow like this:

```
Lower bound: 12
Upper bound: 23
12 13 14 15 16 17 18 19 20 21 22 23
```

5. Modify the previous program in a way that it asks also for the step length:

```
Lower bound: 12
Upper bound: 29
Step size: 5
12 17 22 27
```

   Try this program also giving it a non-positive step size. Try to find out why it locks without reading the next exercise.

6. The previous program locks with non-positive step size because the condition never becomes false. Prevent this. The program should react as follows:

```
Lower bound: 12
Upper bound: 29
Step size: -4
Step size must be positive.
```

7. Compare your program with the following one:

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

int main() {
    cout << "Lower bound: ";
    int lower;
    cin >> lower;
    cout << "Upper bound: ";
    int upper;
    cin >> upper;
    cout << "Step size: ";
    int step;
```

```
        cin >> step;

        if (step > 0) {
            for (int i = lower; i < upper; i = i + step)
                cout << i << " ";
        }
        else
            cout << "Step size must be positive.\n";
    }
```

Chapter 6

Functions

Now we come to one of the most important topics of this tutorial: *functions*. A function is a way to group some code and give it a name; so all the code can later be executed just using its name. Inside a function you can execute ("call") other functions, or even the function itself (in which case the function is called a recursive function).

If a function *takes arguments*, this means that it is possible to pass it one or more objects (the arguments) and that the function will then be able to do things with those arguments. For example, there might be a function called `send` to send emails taking the destination and the text of the mail as arguments. Or a function `power` taking the base and the exponent as arguments, and so on. Notice that not all functions have arguments, and that it is not possible to pass arguments to a function which doesn't take any or not to pass any to one which does; when you write the function, you decide if and what kind of arguments it takes, and after that you have to use it that way.

A function may also have a return value, through which it can pass *one* object back to the caller (the "caller" is the code which called the function). This might be used, for example, in the function `power` described above to give the value of the power back to the caller.

A function consist of two parts: a *declaration*, which explains what the function is called, what types arguments it takes (if any) and what type of return value it has (if any); and the *body*, which is the block of statements executed whenever the function is called. Here an example program:

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

int square(int n) {
    return n * n;
}

int main() {
    int num;
    do {
        cout << "What do you want to know the square "
             << "of (enter 0 to quit): ";
```

```
        cin >> num;
        cout << "The square of " << num << " is "
             << square(num) << ".\n";
    } while (num != 0);
}
```

Does this `square` stuff not look suspiciously like the `int main()` stuff you had to write in each and every program until now without knowing why? Right. But more on that later.

6.1 The Declaration

In the above program, the declaration of the function `square` is

```
int square(int n)
```

What does this mean? Notice that it is divided into three parts. The first part (the `int`) is the *return type*. In this case it means: this function returns an *int*. The second part (`square`) is the *name*. It can be anything you like, with the same limitations as for object names (see 3.2.1). The third part (`(int n)`) is the *argument list*, specifying the type and name of each argument. In this case it means: this function takes one argument of type `int` (which will be called `n` inside the function). Notice that the name of an argument (like "`n`") has *absolutely no influence* of how the caller sees it. That `n` could have been called `x` or `frodo`; for the caller it would not make *any* difference.

To declare a function with no arguments, just leave the parentheses empty.

```
// A silly function which always returns 3
int three() {
    return 3;
}
```

And if you need more than one argument separate them by commas, like here:

```
// Calculates the average of 2 numbers
double average(double n1, double n2) {
    return (n1 + n2) / 2;
}

// Returns the string s n times
// For example mult_string("hello", 3) would return
// "hellohellohello".
string mult_string(string s, int n) {
     string total;
     for (int i = 0; i < n; ++i)
         total = total + s;

     return total;
}
```

6.2 The Body

Now, to the second part: the *body*. It is little more than a normal block of code. One of the differences is that it has access to the arguments, if there are any. For example, in the `square` function described above the first (and only) statement accesses the argument `n`. Of course, if you renamed `n` to `x` or `frodo`, you'd also have to adapt that statement.

6.2.1 `return`

You surely have noticed the *return* statement. It is used — as you probably guessed — to return a value to the caller. That value has of course to be of the same type as the one specified in the declaration; as we declared a function returning an `int`, and as the product of two `int`s is another `int`, our function works fine.

But how does the `return` statement really work? Very simply. As soon as it is executed, the function exits and execution goes on after the function call. The function returns the value passed to the return statement.

Every function with a return value — in other words all functions except the ones described in 6.4 — *must* return a value. Otherwise the compiler wouldn't know what to return to the caller.

6.3 Calling a Function

The function *call* is inside the body of the `main` function, and looks like this:

```
square(num)
```

But what does this do? It calls the function `square`, passing it the value of `num`. As the function returns the square of `n`, it behaves like some kind of variable; we could have obtained the same effect writing

```
int sq = square(num);
cout << sq;
```

There is one thing to think of, though. Whenever we call the function `square` the code in its body is executed; thus

```
cout << square(num);
cout << square(num);
```

and

```
int sq = square(num);
cout << sq;
cout << sq;
```

are *not* the same thing. In the first case the function is called *twice* and its return value is send to standard output; in the second case the function is called *once*, its return value stored in `sq` and then sent to standard output twice. In this case the output is the same, but only because the function `square` depends only on its argument. If we used — for example — a random number function returning a different number each time it would *not* be the same thing.

There are also functions which do not take any arguments at all, like the function `three` shown above. You *must* use the parentheses all the same, like this:

```
cout << three();
```

Only writing `three` instead of `three()` will *not* call the function; it does something completely different which I'll explain later. It is thus possible that your code compiles anyway, but your program will likely not do what you intended.

6.4 `void` Functions

There are also functions with no return value. They are declared just as any other function, but with return type `void`, like this:

```
void display_help() {
    cout << "This program is used to bla bla bla...";
}
```

Of course, you cannot return any value from a void function, but you can use the `return` statement anyway, just with no arguments. It is used to exit the function. It is not *compulsory* that you use `return` though, as it is in the non-void functions; if you don't, the function exits when it reaches the end of the body.

6.5 Scope

Remember what we said about scope referring to the `for` loop (4.4)? Now, with functions, scope becomes even more important, and it is *essential* that you perfectly understand how it works.

As usual, this is best explained with an example. The program

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

// M is a global variable
int m;

int f(int n) {
    // Here a *copy* of the argument is modified, *not*
    // the object which was passed
    n = n + 1;
    return n;
}

int g() {
    // Here the global variable m is increased
    m = m + 1;
    return m;
}
```

```
int main() {
    // This n is local to this function, and *not* the
    // same thing as the n in the function f.
    int n = 5;
    cout << "n = " << n << "\n";
    cout << "f(n) = " << f(n) << "\n";
    cout << "n = " << n << "\n";

    // This is the global m declared at the beginning
    // of the program. It is the same m which is
    // modified in the function g.
    m = 5;
    cout << "m = " << m << "\n";
    cout << "g() = " << g() << "\n";
    cout << "m = " << m << "\n";
}
```

gives the output

```
n = 5
f(n) = 6
n = 5
m = 5
g() = 6
m = 6
```

; as you can see, the *global* variable `m` can be modified by all the functions, while the two *local* variables `n` are completely independent and can be modified only inside the function they are declared in.

6.6 int main()

You have surely noticed that `int main()` is nothing but a function. It is different from other functions, though.

`main` is never called *by your code*. It is called automatically when the program starts. It is *illegal* to call it manually. (Something is said to be *illegal* if it is not allowed by the Standard, or, in other words, if it is an error.)

It is not necessary to put a `return` statement into `main`. In fact, there are only two values you should return from `main`: either `EXIT\_SUCCESS` or `EXIT\_FAILURE`. You return `EXIT\_SUCCESS` if your program terminated successfully, or `EXIT\_FAILURE` if it didn't. (For example, if you write a program to count the words in a text file, it might return `EXIT\_FAILURE` if it cannot open the file.) If you do not return anything, the compiler returns `EXIT\_SUCCESS` for you. You are also allowed to return 0 instead of `EXIT\_SUCCESS`. If you want to use `EXIT\_SUCCESS` and `EXIT\_FAILURE` you must include `<cstdlib>`.

6.7 Pass-by-Value

In C++ arguments are *passed by value*. This means that if you pass a function an argument you do *not* pass it the object itself, only a copy of it. This means that a function cannot modify an argument you gave it. For example, this program

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

void f(int n) {
    n = 4;
}

int main() {
    int m = 1;
    cout << m << "\n";
    f(m);
    cout << m << "\n";
}
```

will print

```
1
1
```

. You have already seen this behavior in the program presented in 6.5; there, `n` was not modified in the function `f`.

6.8 Recursive Functions

It is possible to call a function from itself; in this case the function is *recursive*. For example, the following function is recursive:

```
void f() {
    f();
}
```

Of course, that function would never exit once it started, but continue running until the computer runs out of memory or something else bad happens. A recursive function should thus always have some condition to terminate. Look at the following program which uses a recursive function to print a sequence of numbers:

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

void recursive_function(int start, int end) {
    if (start < end) {
        cout << start << " ";
        recursive_function(start + 1, end);
    }
```

```
}

int main() {
    recursive_function(1, 10);
}
```

It output is

```
1 2 3 4 5 6 7 8 9
```

But how does it work? First, it checks whether `start < end`. If this is true, it prints out start and then calls itself with the same `end` but with `start` increased by one. This means that the following happens:

1. The function is called with `start == 1` and `end == 10`.
2. It checks whether `start < end`. As `start` is 1 and `end` is 10, this is true.
3. It prints out `start`, thus 1.
4. It calls itself with `start + 1` and `end` as arguments. In other words the function returns to step 1, but this time `start` is 2. And then 3. And 4. And so on, until `start` is 10. Then, the check in step 2 fails, and thus the number is *not* printed out and the function is *not* called again, and everything finishes.

This should be quite simple. It starts getting more complicated if we add one line to the function, as follows:

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

void recursive_function(int start, int end) {
    if (start < end) {
        cout << start << " ";
        recursive_function(start + 1, end);
        cout << start << " ";
    }
}

int main() {
    recursive_function(1, 10);
}
```

Now the output is — hold your breath — this one:

```
1 2 3 4 5 6 7 8 9 9 8 7 6 5 4 3 2 1
```

You probably expected something along the lines of `1 1 2 2 3 3 ...`, didn't you? If you knew what'd happen, congratulations. Otherwise, read this explanation.

In a recursive function, at some time there is *more than one* instance of the same function. Each of these instances has its own local variables, which *do not* affect the variables of the other instances. The call

`recursive\_function(start + 1, end)` inside the function returns only when it has finished executing; thus something along the lines of the following happens:

```
recursive_function(1, 10) is called
cout << 1;
  recursive_function(2, 10) is called
  cout << 2;
    recursive_function(3, 10) is called
    cout << 3;
      recursive_function(4, 10) is called
      cout << 4;
        ...
          recursive_function(10, 10) is called
          The condition is false, thus this does NOTHING
        ...
      cout << 4;
      recursive_function(4, 10) exits
    cout << 3;
    recursive_function(3, 10) exits
  cout << 2;
  recursive_function(2, 10) exits
cout << 1;
recursive_function(1, 10) exits
```

Although recursive functions are not too widely used because they are generally error-prone and not very efficient, it is good to know that they exist because some problems can be solved much easier with them than without, especially if speed and stability is not so important.

6.9 Examples

Functions are used by the dozens, hundreds or even thousands in every "real" program; it is thus easy to come up with examples. They are both used to store repeated code and to split functions which would otherwise be too long to understand and maintain.

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

// Print a line of '-' to divide one average from the
// other.
void delimiter() {
    for (int i = 0; i < 79; ++i)
        cout << "-";

    cout << "\n";
}

// Ask for n numbers and calculate their average.
```

```
void average(int n) {
    double sum = 0;

    for (int i = 1; i <= n; ++i) {
        // Ask for a number
        cout << "Enter number " << i << " of " << n
             << ": ";
        double num;
        cin >> num;

        // Add the number to the sum
        sum = sum + num;
    }

    // Print the average
    cout << "The average is: " << sum / n << ".\n";
}

int main() {
    // A for(;;) loop repeats for ever. We use 'return'
    // to jump out of the function directly
    for (;;) {
        cout << "How many numbers do you want to "
             <<"calculate the average of (0 to exit): ";
        int num;
        cin >> num;

        // Jump out of the function if the user entered
        // zero
        if (num == 0)
            return 0;

        // Do the average and display a delimiter (line)
        average(num);
        delimiter();
    }
}
```

6.10 Exercises

1. Write a function called `area_of_circle` taking the radius of a circle as argument and returning its area. Both the radius and the area should be doubles. (The area of a circle is πr^2.) Also write a program to test the function.

2. Write at least two other functions like `area_of_circle` from exercise 1 to calculate other things (area of a square, a triangle, volume of a cube, whatever you like best). Then write a program displaying a menu, asking the user what he wants to calculate, asking for the appropriate data and

doing the right calculation.

3. Take the program from exercise 2 and put the menu in a separate function called `display_menu` which displays the menu, asks the user for his choice and returns an `int` with the user's choice.

4. Modify the function from exercise 3 in a way that it will continue asking until the user inputs a valid selection. (Hint: use a `do...while` loop.)

Chapter 7

Structs

Sometimes it is possible to store all information about something in an already available type; sometimes, it isn't. For example, a number can be easily stored in an `int` or a `double`, and a string can be stored in a `string`. But this is not possible for more complex objects. For example, in 3.1.1 we made the example of a type to store data about wizards in. Of course, there is no such type, and a simple type as `int` or `string` is not sufficient, because a wizard has a name, an age, and many other interesting things which we will not consider for the sake of simplicity.

7.1 `struct`

The simplest way to create a "complex" data type is using a *structure*, or simply *struct* (after the keyword `struct` used to declare one). Strictly speaking, in C++ it is not possible to create a new type from the ground up, but only to group already existing types.

If we want to create a `Wizard` data type, first we must think about what we want to store in it. To keep it simple, we will only store the wizard's name and his age. As we can use only already existing types, we'll take a `string` for the name and an `int` for the age.

Suppose we want three wizard objects: Gandalf, Sauron and Saruman. *Without* introducing a new data type, we'd do it somehow like this:

```
string gandalf_name;
int gandalf_age;

string sauron_name;
int sauron_age;

string saruman_name;
int saruman_age;
```

This is of course very tedious. And if we wanted to add some other data to each wizard — for example his height — we'd have to add it manually to each of them. This is of course no way to go. And that's why there are structs.

```
struct Wizard {
```

```
    string name;
    string age;
};
```

This defines the new type `Wizard`, which is a type just as `int` or `string`. Thus, we can declare objects of this new type, like this:

```
Wizard gandalf, sauron, saruman;
```

Notice that it is a good idea to use some convention to distinguish *types* from *objects*; I will always write objects `like_this` and types `LikeThis`. This has the advantage that it is possible to declare an object with the same name as a type, which would otherwise not be possible. (For example, if we had called our new type `wizard` instead of `Wizard`, we could not declare an *object* named `wizard` too.)

But what can we do with these objects? Not very much, indeed. We cannot assign to them, we cannot compare them, we cannot print them and we cannot read data into them from the console. We can do only two things: declaring them and accessing their members.

7.1.1 Member Access

A *member* of a struct is one of the objects declared inside it. The members of `Wizard` are `name` and `age`. They can be accessed using the dot operator (`object.member`), as follows:

```
gandalf.name = "Gandalf";
gandalf.age = 123;
cout << gandalf.name << " is " << gandalf.age << " years old.\n";
```

An object inside a struct is treated just as any other object; you can do exactly the same things with it as you can with a "normal" one.

Here a complete program doing all the things I've explained until now:

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

int main() {
    struct Wizard {
        string name;
        int age;
    };

    Wizard gandalf;
    gandalf.name = "Gandalf";
    gandalf.age = 123;

    cout << gandalf.name << " is " << gandalf.age << " years old.\n";
}
```

7.1.2 Helper Functions

In the previous example you might have wondered about why these structs are so important; in fact, you might have written the same program without them, and it would even have been a bit shorter.

The magic of objects, in fact, isn't that you can group smaller entities into new types and then access these entities using the dot operator. Its magic is *modularization*. This means that it is easy to split a program into different parts (called *modules*) which are more or less independent of each other.

Modularization is no simple business; one step towards it, though, are helper functions: functions which deal with a certain type of object. For example, we might want to print all we know about a wizard on the screen. One way is doing it the way we've done it above; another — better — way is to put all the printing into a separate function, and call it whenever we need it, like this:

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

struct Wizard {
    string name;
    int age;
};

void wizard_print(Wizard wizard) {
    cout << wizard.name << " is " << wizard.age << " years old.\n";
}

int main() {
    Wizard gandalf, sauron, saruman;

    gandalf.name = "Gandalf";
    gandalf.age = 123;

    sauron.name = "Sauron";
    sauron.age = 234;

    saruman.name = "Saruman";
    saruman.age = 345;

    wizard_print(gandalf);
    wizard_print(sauron);
    wizard_print(saruman);
}
```

As you can see, *now* it makes much more sense. If, for example, we want to print `Gandalf, 123` and so on instead of `Gandalf is 123 years old.` we only need to change *one* function, and can leave the rest alone.

It would be nice if we could write functions to *change* a `Wizard` as well (like for example `input_wizard` to ask for a wizard's name and age) but for the time

being we cannot do that. As I pointed out in 6.7, function arguments are passed by value; thus if we wrote a function like this:

```
void input_wizard(Wizard wizard) {
    cout << "Please enter the name of the wizard: ";
    getline(cin, wizard.name);
    cout << "Please enter " << wizard.name << "'s age: ";
    cin >> wizard.age;
}
```

would not work as expected. If you called it like this:

```
input_wizard(gandalf);
```

the compiler would make a *copy* of `gandalf` and pass it to the function. The function would then read in the data into the copy, without modifying the original. We'll talk about that again in 8.

Chapter 8

References

A *reference* is a special object which refers to another, "normal", object. In other words it is an object which has its own name and type, but "uses" the data of another object (which must be of the same type, of course). It is similar to the links to files which certain OSs let you create: they have a name of their own, but use the data of some other file. A reference is declared like this:

```
int i;      //i is a normal int
int& r = i; //r is a reference to i
```

(Notice the `&` after the type name.)

You *must* initialize a reference when you declare it, because otherwise it wouldn't have an object to "use".

In their simplest form, references are just a new name for an already existing object, like here:

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

int main() {
    int i = 5;
    int& r = i; // Declare a reference to i
    cout << i << "\n";
    cout << r << "\n";

    // Now modify i
    i = 7;
    cout << i << "\n";
    cout << r << "\n";

    // Now modify i through r
    r = 9;
    cout << i << "\n";
    cout << r << "\n";
}
```

The output of this program is

```
5
5
7
7
9
9
```

8.1 Pass-by-Reference

Of course, used that way references are perfectly useless. In fact, that's not how they are generally used. As a reference is *not* just an alias for some existing object, but object of its own, which *refers* to another object, it can be used in many other ways. As you can modify the original object through a reference, it is possible to pass a reference to some object and *not* the object itself to a function, which is thus able to modify the original object, just like here:

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

void increment(int& i) {
    ++i; // Remember that ++i is a short form for i = i + 1
}

int main() {
    int test = 10;
    cout << test << "\n";
    increment(test);
    cout << test << "\n";
}
```

The output of this program is

```
10
11
```

This method to pass arguments is called *passing by reference*, opposed to *passing by value* as described in 6.7.

Of course, you can not pass a constant value by reference; in other words, something like

```
increment(13);
```

is illegal because it is of course not possible to increment `13`. You can only pass variables like this.

Passing by reference is especially useful for `structs`; it is the simplest (but not only) way to overcome the limitation explained in 7.1.2. For example, now we can write a function to read in a `Wizard` object from standard input:

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

```
struct Wizard {
    string name;
    int age;
};

// Notice how we use references to be able to modify
// the wizard inside the function
void wizard_input(Wizard& wizard) {
    cout << "Name: ";
    cin >> wizard.name;
    cout << "Age: ";
    cin >> wizard.age;
}

// Here we pass by value and not by reference because
// we don't need to modify anything
void wizard_output(Wizard wizard) {
    cout << wizard.name << " is " << wizard.age << ".\n";
}

int main() {
    Wizard wizard;
    cout << "Please tell me about your wizard:\n";
    wizard_input(wizard);
    wizard_output(wizard);
}
```

This program works like this:

```
Please tell me about your wizard:
Name: Gandalf
Age: 123
Gandalf is 123.
```

8.1.1 Speed

If you don't want to read this now, or don't understand it, feel free skipping it. You should only know that it is useful to pass `structs` by reference even if you don't want to modify the object you're passing, because it is generally faster.

Passing by reference is not only useful to pass arguments to functions which need to modify them; it may also be faster and require less memory than passing by value.

A reference generally has the size of a machine word; this means that on a 32-bit processor (like all Pentiums, from I to IV, for example) it is 32 bits wide, on a 64 bit processor 64 bits wide and so on. Then, arguments passed by reference need to be *dereferenced* by the compiler while arguments passed by value don't; this makes references slower. But if you pass a reference the compiler does not need to copy the argument, which is, of course, and advantage.

As a rule of thumb you can say that a Plain Old Data (POD) (explanation follows) is faster if passed by value, while a complex type like a `struct` — which is generally also quite a lot bigger than a POD — is faster if passed by reference.

A POD is a so-called *built-in type*, like `int` and `double`. Notice, though, that `string` — and many others — are part of C++, but not PODs anyway. As another rule of thumb (yes, I love them) you can say that the types you can use even without including any headers are PODs, while the others aren't. This isn't always true, though, and that's why it is a rule of thumb.

8.2 Constants

In some cases you want to pass an argument by reference even if you don't want to modify it — be it for the sake of speed (8.1.1), memory (8.1.1, too) or some other reason. It would be nice if you could tell the compiler not to allow that object to be modified. That's what the `const` keyword is here for. But before introducing how to use it on references, let's use it on plain objects.

8.2.1 Constant Objects

Sometimes it is necessary to use the same constant — for example the number 20 — in several places in a program, as here:

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

int main() {
    cout << "Countdown will last 20 minutes.\n";
    for (int i = 20; i > 0; --i)
        cout << i << " ";

    cout << "GO!\n";
}
```

This is of course very ugly. If you wanted to modify the program to use 10 instead of 20, you'd have to change it in more than one place. This is not only tedious, but also dangerous. You might forget one, and make the program buggy. It might seem a good idea to replace the constant with a variable, and to initialize the variable at the very beginning, like this:

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

int main() {
    int count = 20;

    cout << "Countdown will last " << count << " minutes.\n";
    for (int i = count; i > 0; --i)
        cout << i << " ";

    cout << "GO!\n";
}
```

This is of course already a lot better. There is still a problem, though. You might modify the variable by mistake, like here:

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

int main() {
    int count = 20;

    cout << "Countdown will last " << count << " minutes.\n";
    count = 13; // OOPS!
    for (int i = count; i > 0; --i)
        cout << i << " ";

    cout << "GO!\n";
}
```

Of course, the program wouldn't work correctly anymore.

What we need is a variable which cannot be modified. That's exactly what a constant is. The main differences between variables and constants are:

- A constant must be initialized.
- A constant cannot be changed.

Using constants, our nice countdown program looks like this:

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

int main() {
    const int count = 20;

    cout << "Countdown will last " << count << " minutes.\n";
    for (int i = count; i > 0; --i)
        cout << i << " ";

    cout << "GO!\n";
}
```

If we would introduce the evil `count = 13` again, it would not compile, because a constant cannot be modified. For the same reason you cannot even pass a constant to the `increment()` function described above.

8.2.2 Constant References

We have already seen how references can be used to pass an argument which is modified in the function, like this:

```
void decrement(int& i) {
    --i;
}
```

Obviously, it is legal to do something like this:

```
int x = 3;
decrement(x);
```

And obviously it is illegal to do something like this:

```
decrement(3);
```

because `3`, being a number, cannot be modified. For the same reason you cannot pass a `const int` to `decrement`, like here:

```
const int x = 3;
decrement(3);
```

If you could do that — modifying a constant — `const` would be totally useless.

For the reasons explained in 8.1.1, it might be useful sometimes to pass an argument by reference and not by value even if the function does not modify it. Consider this example:

```
int triple(int& n) {
    return n * 3;
}
```

(Of course, this function is not very useful, and we could have used `int` instead of `int&` without any problems. Please bear with it as an example until I find something better.)

As one would expect, it is perfectly legal to use it as follows:

```
int x = 5;
cout << triple(x); // Prints '15'
```

But the following use, which is just as sensible, is illegal:

```
cout << triple(5); // ERROR
```

This apparently "strange" behavior is not strange at all. The function `triple` takes a reference to an `int`, thus it *could* modify it. It doesn't matter if it does or not; the fact that it could is enough for the compiler to refuse to take a constant (a literal number like `5` is constant) instead of a variable.

If you want to write a function taking a reference to something as a parameter, but you do not modify the passed object through the reference, you simply have to declare the reference as `const`, as follows:

```
int triple(const int& n) {
    return n * 3;
}
```

With this new function `triple(5)` is completely legal.

As already pointed out in 8.1.1, references are generally faster if used with user defined types; our user defined type *par excellence* is `Wizard`, and thus we'll write a function taking a const reference to a `Wizard`.

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

using namespace std;

struct Wizard {
    string name;
```

```
    int age;
};

void read_wizard(Wizard& wizard) {
    cout << "Name: ";
    cin >> wizard.name;
    cout << "Age: ";
    cin >> wizard.age;
}

void print_wizard(const Wizard& wizard) {
    cout << wizard.name << " is " << wizard.age << " years old.\n";
}

int main() {
    Wizard wizard;
    wizard_read(wizard);
    wizard_print(wizard);
}
```

It is, as already said, not possible to modify an object through a const reference; thus, something like

```
wizard.age = 100;
```

would be legal inside `read_wizard` but not inside `print_wizard`.

For the same reason, you cannot pass a const reference to a function which expects a non-const reference; in other words, you cannot call

```
read_wizard(wizard);
```

from `print_wizard`, but you can call

```
print_wizard(wizard);
```

from `read_wizard`.

Chapter 9

Classes

As you have seen in 7.1, structs are a very nice way to create new data types. And as you have seen in 7.1.2, it is possible to write functions like `read_wizard` and `print_wizard` to access them in a simple way.

Fortunately there is also another way to create new types in C++, which is much more powerful and simple: classes.

A class is declared just like a struct; thus, we could declare `Wizard` as follows:

```
class Wizard {
    string name;
    int age;
};
```

Notice that the semicolon after the closing brace is, just like in a `struct`, *not* an optional.

9.1 Member Scope

There is one big difference between the struct `Wizard` and the class `Wizard`: the scope of the members. While in the struct — which has so-called *public scope* — one could access the members using the dot operator (`wizard.age`, for example), this is not possible in the class, which has *private scope*. In fact, it is not possible to access the members *at all* from outside the class.

The problem is that each member of a struct or a class has either public, private or protected scope. The public members can be accessed from anywhere. The private members can be accessed only from inside the class. The protected members are somehow in between, but we'll talk about them later; you don't need them yet.

By default (if you don't specify anything else) the members of a class are private. If you wanted the class `Wizard` to be perfectly analogous to the struct, you'd have to declare it like this:

```
class Wizard {
public: // Whatever follows this is public
    string name;
    int age;
};
```

If you wanted to declare one member private and one public, you could do it like this:

```
class Wizard {
public:
    string name;
private:
    int age;
};
```

or, using the fact that the default is private, simply like this:

```
class Wizard {
    int age;
public:
    string name;
};
```

9.2 Member Functions

Let's go back to our good old `Wizard` now. Its original declaration was

```
struct Wizard {
    string name;
    int age;
};
```

As it is a struct, all members are public. This means that every programmer getting his hands on a `Wizard` object can change its age and name as it likes. This is of course a Bad Thing. To avoid that, we could declare `Wizard` as a class, like this:

```
class Wizard {
    string name;
    int age;
};
```

This way, all members being private, no function could mess with our poor wizard. The problem is that no function could access any of its data either; in fact, it wouldn't even be possible to give the name and the age meaningful values.

What we need are some privileged functions which can access the wizards interior; we know those functions won't harm our wizard, and all the other programmers could use those functions to do whatever they need to do with the wizard.

That's exactly what member functions are. They are functions which *belong* to the class, and can thus access all of its members — public, private and protected — without any restrictions. The `read_wizard` function could be easily implemented as a member function as follows (we call it only `read` because, being part of `Wizard`, it is obvious that it refers to wizards and not to witches):

```
class Wizard {
    // The default is private, so we don't need to
    // specify
    string name;
    int age;
public:
    void read() {
        cout << "Name: ";
        cin >> name;
        cout << "Age: ";
        cin >> age;
    }
};
```

Before you start wondering about why `read` does not specify *which* wizard `name` and `age` refer to, look at how member functions are called:

```
Wizard gandalf;
gandalf.read();
```

Now it should be clear. The members `name` and `age` used in `read` automatically refer to `gandalf.name` and `gandalf.age` because `read` was called through `gandalf`. If instead of `gandalf.read()` you had called `sauron.read()` they would have referred to `sauron.name` and `sauron.age`.

9.2.1 Constant Member Functions

In 8.2.2 I pointed out why it is important to use a const reference (instead of a non-const one) when a function does not modify an argument. With member functions the same problem arises: some of them, as `read()` may modify their object, others, as for example `print()`, do not.

As we do not explicitly specify the reference to the object which is modified, it is not possible to use the same method anymore. Instead of that, the `const` keyword is appended *after* the closing parenthesis of the function declaration, like this:

```
class SomeClass {
    // ...
    void this_function_does_not_modify_the_object() const {
        // ...
    }
    void this_function_modifies_the_object() {
        // ...
    }
};
```

You cannot modify the members of an object in a const member function, but they are the only type of member function you can call on a constant object. If `print` is a const member function and `read` isn't, you can call `print` on a constant object but not `read`.

This means that if, for example, you have a function `f` taking a const reference to a wizard, you can only call the const member functions of that wizard inside the function, as follows:

```
#include <iostream> // for cin and cout
#include <string>
using namespace std;

class Wizard {
public:
    void read() {
        cout << "Name: ";
        cin >> name;
        cout << "Age: ";
        cin >> age;
    }
    void print() const {
        cout << "Name: " << name << "\nAge: " << age
             << '\n';
    }
private:
    string name;
    int age;
};

void f(const Wizard& wizard) {
    wizard.print(); // This is ok

    // wizard.read();  // ERROR!
                       // You cannot modify wizard!
}

int main() {
    Wizard some_wizard;
    some_wizard.read();
    f(some_wizard);
}
```

9.2.2 Const Correctness

Obviously, it would not be possible to declare `Wizard::read()` as const because it modifies the `Wizard` object. (Notice how I wrote `Wizard::read()`. In C++, that's the way to refer to a member function; it means "the `read()` member function of the `Wizard` class"). But it would be possible to define `Wizard::print()` as non-const, even if it does not modify the `Wizard` object. If you did so, you could not even *print* a constant `Wizard`; in other words, even the line `wizard.print()` in the example program above would be an error.

Declaring functions which do not modify the object as const is thus important; if you don't, you won't be able to use them in const objects. This is known as *const correctness*, and it is much more important than you might think right now.

9.3 Encapsulation

The technique of declaring the member functions as public and the member variables as private, thus not allowing anybody to mess with them directly, is called *encapsulation*. Generally it is a sign of a good design; this does of course not mean that any public member variable is bad and should be wrapped into member functions. There are cases where that wouldn't make any sense.

Declaring member functions to access the content of a class has many advantages; one of them is that it is easy to change the way the data is actually stored inside the class; if you allow everyone to access the member variables directly, this is not possible any more. It may seem more work at the beginning, but trust me, in the end you'll be glad if you did so.

For example, imagine a word processor written in C++, which has class `Document` storing, well, documents. Imagine that the `Document` class has a member variable of type `string` called `text` to store the text, and some other member variables to store all the things which the text editor must know about the text (like the author, the creation date, and so on). Suppose that the `Document` class has also a member function called `save()`.

Now, if you have not encapsulate the variables, everytime `save()` is called you have to write the whole document to the hard disk, even if it was not modified because you cannot easily know whether it was changed or not. After all, any part of the outside code could have changed something. (It would of course be possible to calculate some kind of checksum everytime the document was saved, and recalculating it when `save()` is called and comparing it to the previous checksum, but that would just be overkill).

If you have encapsulated the variables, everything becomes much easier. For example, you have a const member function called `get_author()` to retrieve the author's name, and one (non-const, of course) called `set_author()` to change it. All you have to do now is adding a single variable of type `bool` to the class, called `changed`. (A `bool` variable can only have two values: `true` or `false`). Now, if `set_author()` (or any other member function modifying the class) is called you set the `changed` to `true`; so in `save()` you can easily check whether the document was changed.

```
void Document::save() {
    if (changed) { // equivalent to "if (changed == true)"
        // save the document
        // ...
        changed = false;
    }
}
```

Or, another more realistic example. Suppose you have written a very complex algorithm to do special searches on a list of objects, which is represented by an instance of the class `List`, which stores all objects as an array accessible through `List::objects`. Your algorithm will use statements such as `some_list.objects[n]` to access the different objects. But as the number of objects grows larger and larger, you realize that it is not possible to keep all of them in memory at the same time. Thus you decide to change the implementation of `List` so that it only keeps a small set of objects in memory, and leaves the rest on disk, and automatically loads the necessary objects if the algorith

requests them. Now you have a problem, because there is *no way* you can do that without changing the interface of `List`. You'll have to adapt or at least recompile the algorithm. If you had used proper encapsulation and defined a function such as `List::get_object(int index)` to retrieve the objects, you could just change its implementation without touching the algorithm.

9.4 Structs vs Classes

You might wonder now why I even bothered introducing structs, if everything a struct can do can be achieved through a class with a `public:` at the beginning.

In C++ the difference between a struct and a class is only the default scope of the members: public in structs and private in classes. They are interchangeable for all the rest: structs can have member functions and everything else a class can.

The programming language C had only structs and no classes. Those structs did not allow member functions and scope declarations; they were used only as described in 7.1. C++, which was initially only an enhanced C, kept the structs and introduced the classes, which were, at least at the beginning, not much different from the structs. Classes evolved to what they are now, and, as it was easiest for the compiler writers, `struct` became an alias for `class`, with the default scope as only difference.

Of course you can do what you like, but I'd recommend you to follow the rest of the world and use structs when you use none of the features of C++ features (as scope, and member functions), and classes for everything else. In other words structs are only for plain variable groups with no member functions.

Chapter 10

Pointers

Pointers are a very complex but incredibly powerful concept. They allow you to write elegant and efficient code, but also to introduce bugs which are very difficult to trace and fix.

Every byte which is stored in a computer's RAM has an address. Simplifying, you could say that the first byte in the RAM has the address 0, the second the address 1 and so on, as if all the RAM were a big array of bytes. (In reality, memory management a very complex subject; a good introduction would be longer than this entire tutorial. But for our needs this simplification is good enough.)

A pointer is exactly such an address. Giving it different adresses you can point it to different parts of memory; that's why it is called a pointer.

The *value* of a pointer is such an address; as it may be completely different on different machines and between different runs of the same program, it is completely useless and you should never use it. In fact, a pointer *contains no data*, in only *point to data*. The interesting part is *what the pointer is pointing to*, i.e. the data which resides at the address specified by the pointer. But let's look at some examples now.

10.1 Using Pointers

10.1.1 Declaring Pointers

In C++ (and also in C) pointers are a special type of variable, and must be declared just like any other variable. A pointer has the type "pointer to something", where something is any other data type. (You can even declare pointers to pointers, but more on that later).

For example, to declare a pointer to `char` you'd do something like this:

```
char* p;
```

The asterisk "`*`" means that `p` is a pointer; analogously a pointer to `int` is declared as follows:

```
int* p;
```

Notice that the space after the asterisk is optional; in fact, the following ways to declare a pointer are all equivalent, and only depend on the programmer's favourite style:

```
int* p;
int*p;
int *p;
```

There's a pitfall though. With "normal" (non-pointer) data types, you can declare more than one variable at once:

10.1.2 Assigning Pointers

Of course, to do anything useful with pointers you first need to assign them a value. You can either assign it a value from another pointer, or put the address of a real object into it:

```
// declare two pointers to int p and q
int* p, *q;
% todo
```

10.1.3 Dereferencing Pointers

10.1.4 The `NULL` Pointer

10.1.5 A Simple Example

10.2 Pointer Arithmetics

10.3 Examples

10.4 Exercises

Acronyms

CPU Central Processing Unit

The most important part of a modern computer.

OOP Object-Oriented Programming

A programming style which deals with objects instead of more traditional entities like functions. The functions are associated to the object, and not vice versa. In other words, the object is in the foreground and not the functions anymore.

OS Operating System

You know what an OS is, don't you?

POD Plain Old Data

A POD type is a data type built into the C++ language, *not* one defined in the standard library or by your program. Examples of PODs include `int`, `double`, `size_t` and many others. A type defined as a POD using `typedef` is also a POD.

RAM Random Access Memory

The volatile memory in a computer, used to store temporary data. Whatever is stored in it is lost forever once the power is turned off.

Bibliography

[LOTR] J. R. R. Tolkien, *Lord of the Rings*.
Various publishers and editions.

[TCPL] Bjarne Stroustrup, *The C++ Programming Language, Special Edition*.
Addison-Wesley, ISBN 0-201-70073-5.

Bjarne Stroustrup, *The C++ Programming Language, 3rd Edition*.
Addison-Wesley, ISBN ?-???-?????-?.