This document is part of the HTML publication "**An Introduction to the Imperative Part of C++**"

The original version was produced by **Rob Miller** at **Imperial College London**, September 1996.

Version 1.1 (modified by **David Clark** at **Imperial College London**, September 1997)

Version 1.2 (modified by **Bob White** at **Imperial College London**, September 1998)

Version 1.3, 1.4, 2.0, ..., 2.15 (modified by **William Knottenbelt** at **Imperial College London**, September 1999-September 2016)

1. Introducing C++

1.1 Some Remarks about Programming

Programming is a core activity in the process of performing tasks or solving problems with the aid of a computer. An idealised picture is:

[problem or task specification] - **COMPUTER** - [solution or completed task]

Unfortunately things are not (yet) that simple. In particular, the "specification" cannot be given to the computer using natural language. Moreover, it cannot (yet) just be a description of the problem or task, but has to contain information about how the problem is to be solved or the task is to be executed. Hence we need programming languages. Click here for a more detailed view of the problem solving pipeline.

There are many different programming languages, and many ways to classify them. For example, "high-level" programming languages are languages whose syntax is relatively close to natural language, whereas the syntax of "low-level" languages includes many technical references to the nuts and bolts (0's and 1's, etc.) of the computer. "Declarative" languages (as opposed to "imperative" or "procedural" languages) enable the programmer to minimise his or her account of *how* the computer is to solve a problem or produce a particular output. "Object-oriented languages" reflect a particular way of thinking about problems and tasks in terms of identifying and describing the behaviour of the relevant "objects". *Smalltalk* is an example of a pure object-oriented language. C++ includes facilities for object-oriented programming, as well as for more conventional procedural programming.

Proponents of different languages and styles of languages sometimes make extravagant claims. For example, it is sometimes claimed that (well written) object-oriented programs reflect the way in which humans think about solving problems. Judge for yourselves!

(BACK TO COURSE CONTENTS)

1.2 The Origins of C++

C++ was developed by Bjarne Stroustrup of AT&T Bell Laboratories in the early 1980's, and is based on the C language. The name is a pun - "++" is a syntactic construct used in C (to increment a variable), and C++ is intended as an incremental improvement of C. Most of C is a subset of C++, so that most C programs can be compiled (i.e. converted into a series of low-level instructions that the computer can execute directly) using a C++ compiler.

C is in many ways hard to categorise. Compared to assembly language it is high-level, but it nevertheless includes many low-level facilities to directly manipulate the computer's memory. It is therefore an excellent language for writing efficient "systems" programs. But for other types of programs, C code can be hard to understand, and C programs can therefore be particularly prone to certain types of error. The extra object-oriented facilities in C++ are partly included to overcome these shortcomings.

(BACK TO COURSE CONTENTS)

1.3 ANSI/ISO C++

The American National Standards Institution (ANSI) and the International Standards Organisation (ISO) provide "official" and generally accepted standard definitions of many programming languages, including C and C++. Such standards are important. A program written only in ANSI/ISO C++ is guaranteed to run on any computer whose supporting software conforms to the the standard. In other words, the standard guarantees that standard-compliant C++ programs are portable. In practice most versions of C++ include ANSI/ISO C++ as a core language, but also include extra machine-dependent features to allow smooth interaction with different computers' operating systems. These machine dependent features should be used sparingly. Moreover, when parts of a C++ program use non-compliant components of the language, these should be clearly marked, and as far a possible separated from the rest of the program, so as to make modification of the program for different machines and operating systems as easy as possible. The four most significant revisions of the C++ standard are C++98 (1998), C++03 (2003) and C++11 (2011) and C++14 (2014). The next iteration is currently expected in 2017. Of course, it can be a challenging task for software engineers, compiler writers and lecturers (!) to keep track of all the revisions that appear in each major version of the standard. You may wish to read further about the ongoing efforts that are being made to make the widely-used g++ compiler compliant with the C++11 standard and compliant with the C++14 standard.

(BACK TO COURSE CONTENTS)

1.4 The C++ Programming Environment in UNIX

The best way to learn a programming language is to try writing programs and test them on a computer! To do this, we need several pieces of software:

- An editor with which to write and modify the C++ program components or source code,
- A compiler with which to convert the source code into machine instructions which can be executed by the computer directly,
- A linking program with which to link the compiled program components with each other and with a selection of routines from existing libraries of computer code, in order to form the complete machine-executable object program,
- A debugger to help diagnose problems, either in compiling programs in the first place, or if the object program runs but gives unintended results.

There are several editors available for UNIX-based systems. Two of the most popular editors are emacs and vi. For the compiler and linker, we will be using the GNU g++ compiler/linker, and for the debugger we will be using the GNU debugger gdb. For those that prefer an integrated development environment (IDE) that combines an editor, a compiler, a linking program and a debugger in a single programming environment (in a similar way to Microsoft Developer Studio under Windows NT), there are also IDEs available for UNIX (e.g. Eclipse, xcode, kdevelop etc.)

Appendix A.1 of these notes is a brief operational guide to emacs and g++ so that you can get going without delay. If you are interested in learning more about UNIX you can click here.

(BACK TO COURSE CONTENTS)

1.5 An Example C++ Program

Here is an example of a complete C++ program:

```
// The C++ compiler ignores comments which start with
// double slashes like this, up to the end of the line.

/* Comments can also be written starting with a slash
 followed by a star, and ending with a star followed by
 a slash. As you can see, comments written in this way
 can span more than one line. */

/* Programs should ALWAYS include plenty of comments! */

/* Author: Rob Miller and William Knottenbelt
 Program last changed: 30th September 2001 */

/* This program prompts the user for the current year, the user's
current age, and another year. It then calculates the age
that the user was or will be in the second year entered. */

#include <iostream>

using namespace std;

int main()
{
        int year_now, age_now, another_year, another_age;

        cout << "Enter the current year then press RETURN.\n";
        cin >> year_now;

        cout << "Enter your current age in years.\n";
        cin >> age_now;

        cout << "Enter the year for which you wish to know your age.\n";
        cin >> another_year;

        another_age = another_year - (year_now - age_now);

        if (another_age >= 0) {
                cout << "Your age in " << another_year << ": ";
                cout << another_age << "\n";
        } else {
                cout << "You weren't even born in ";
                cout << another_year << "!\n";
        }

        return 0;
}
```

Program 1.5.1

This program illustrates several general features of all C++ programs. It begins (after the comment lines) with the statement

```
#include <iostream>
```

This statement is called an include directive. It tells the compiler and the linker that the program will need to be linked to a library of routines that handle input from the keyboard and output to the screen (specifically the `cin` and `cout` statements that appear later). The header file "iostream" contains basic information about this library. You will learn much more about libraries of code later in this course.

After the include directive is the line:

```
using namespace std;
```

This statement is called a *using* directive. The latest versions of the C++ standard divide names (e.g. `cin` and `cout`) into subcollections of names called *namespaces*. This particular *using* directive says the program will be using names that have a meaning defined for them in the `std` namespace (in this case the `iostream` header defines meanings for `cout` and `cin` in the `std` namespace).

Some older C++ compilers do not support namespaces. In this case you can use the older form of the include directive (that does not require a *using* directive, and places all names in a single global namespace):

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

Some of the legacy code you encounter in industry may be written using this older style for headers.

Because the program is short, it is easily packaged up into a single list of program statements and commands. After the include and using directives, the basic structure of the program is:

```
int main()
{
        First statement;
        ...
        ...
        Last statement;

        return 0;
}
```

All C++ programs have this basic "top-level" structure. Notice that each statement in the body of the program ends with a semicolon. In a well-designed large program, many of these statements will include references or calls to sub-programs, listed after the main program or in a separate file. These sub-programs have roughly the same outline structure as the program here, but there is always exactly one such structure called `main`. Again, you will learn more about sub-programs later in the course.

When at the end of the main program, the line

```
return 0;
```

means "return the value 0 to the computer's operating system to signal that the program has completed successfully". More generally, *return statements* signal that the particular sub-program has finished, and return a value, along with the flow of control, to the program level above. More about this later.

Our example program uses four *variables*:

`year_now`, `age_now`, `another_year` and `another_age`

Program variables are not like variables in mathematics. They are more like symbolic names for "pockets of computer memory" which can be used to store different values at different times during the program execution. These variables are first introduced in our program in the variable declaration

```
int year_now, age_now, another_year, another_age;
```

which signals to the compiler that it should set aside enough memory to store four variables of type "`int`" (integer) during the rest of the program execution. Hence variables should always be declared before being used in a program. Indeed, it is considered good style and practice to declare all the variables to be used in a program or sub-program at the beginning. Variables can be one of several different types in C++, and we will discuss variables and types at some length later.

(BACK TO COURSE CONTENTS)

1.6 Very Simple Input, Output and Assignment

After we have compiled the program above, we can *run* it. The result will be something like

```
Enter current year then press RETURN.
1996
Enter your current age in years.
36
Enter the year for which you wish to know your age.
2011
Your age in 2011: 51
```

The first, third, fifth and seventh lines above are produced on the screen by the program. In general, the program statement

```
cout << Expression1 << Expression2 << ... << ExpressionN;
```

will produce the screen output

```
Expression1Expression2...ExpressionN
```

The series of statements

```
cout << Expression1;
cout << Expression2;
...
...
cout << ExpressionN;
```

will produce an identical output. If spaces or new lines are needed between the output expressions, these have to be included explicitly, with a `" "` or a `"\n"` respectively. The expression `endl` can also be used to output a new line, and in many cases is preferable to using `"\n"` since it has the side-effect of flushing the output buffer (output is often stored internally and printed in chunks when sufficient output has been accumulated; using `endl` forces all output to appear on the screen immediately).

The numbers in **`bold`** in the example screen output above have been typed in by the user. In this particular program run, the program statement

```
cin >> year_now;
```

has resulted in the variable `year_now` being *assigned* the value 2001 at the point when the user pressed RETURN after typing in "2001". Programs can also include assignment statements, a

simple example of which is the statement

```
another_age = another_year - (year_now - age_now);
```

Hence the symbol = means "is assigned the value of". ("Equals" is represented in C++ as ==.)

(BACK TO COURSE CONTENTS)

1.7 Simple Flow of Control

The last few lines of our example program (other than "`return 0`") are:

```
if (another_age >= 0) {
        cout << "Your age in " << another_year << ": ";
        cout << another_age << "\n";
}
else {
        cout << "You weren't even born in ";
        cout << another_year << "!\n";
}
```

The "if ... else ..." branching mechanism is a familiar construct in many procedural programming languages. In C++, it is simply called an *if statement*, and the general syntax is

```
if (condition) {
        Statement1;
        ...
        ...
        StatementN;
} else {
        StatementN+1;
        ...
        ...
        StatementN+M;
}
```

The "else" part of an "if statement" may be omitted, and furthermore, if there is just one *Statement* after the "if (condition)", it may be simply written as

```
if (condition)
        Statement;
```

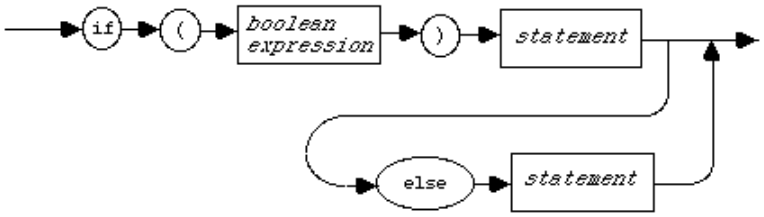
It is quite common to find "if statements" strung together in programs, as follows:

```
...
...
if (total_test_score < 50)
        cout << "You are a failure. You must study much harder.\n";
else if (total_test_score < 65)
        cout << "You have just scraped through the test.\n";
else if (total_test_score < 80)
        cout << "You have done quite well.\n";
else if (total_test_score < 95)
        cout << "Your score is excellent. Well done.\n";
else {
        cout << "You cheated!\n";
        total_test_score = 0;
}
...
...
```

This program fragment has quite a complicated logical structure, but we can confirm that it is legal in C++ by referring to the syntax diagram for "if statements". In such diagrams, the terms

enclosed in ovals or circles refer to program components that literally appear in programs. Terms enclosed in boxes refer to program components that require further definition, perhaps with another syntax diagram. A collection of such diagrams can serve as a formal definition of a programming language's syntax (although they do not help distinguish between good and bad programming style!).

Below is the syntax diagram for an "if statement". It is best understood in conjunction with the syntax diagram for a "statement". In particular, notice that the diagram doesn't explicitly include the ";" or "{}" delimiters, since these are built into the definition (syntax diagram) of "statement".



1.7.1 Syntax diagram for an If Statement

The C++ compiler accepts the program fragment in our example by counting all of the **bold** text in

```
...
...
if (total_test_score < 50)
        cout << "You are a failure. You must study much harder.\n";
else if (total_test_score < 65)
        cout << "You have just scraped through the test.\n";
else if (total_test_score < 80)
        cout << "You have done quite well.\n";
else if (total_test_score < 95)
        cout << "Your score is excellent. Well done.\n";
else {
        cout << "You cheated!\n";
        total_test_score = 0;
}
...
...
```

as the single statement which must follow the first else.

(BACK TO COURSE CONTENTS)

1.8 Preliminary Remarks about Program Style

As far as the C++ compiler is concerned, the following program is exactly the same as the program in Section 1.5:

```
#include <iostream> using namespace std; int main()
 { int year_now, age_now, another_year, another_age; cout <<
 "Enter the current year then press RETURN.\n"; cin >> year_now; cout
 << "Enter your current age in years.\n"; cin >> age_now; cout <<
 "Enter the year for which you wish to know your age.\n"; cin >>
 another_year; another_age = another_year - (year_now - age_now); if
 (another_age >= 0) { cout << "Your age in " << another_year << ": ";
 cout << another_age << "\n"; } else { cout <<
 "You weren't even born in "; cout << another_year << "!\n"; } return
 0; }
```

However, the lack of *program comments*, *spaces*, *new lines* and *indentation* makes this program unacceptable. There is much more to developing a good programming style than learning to lay out programs properly, but it is a good start! Be consistent with your program layout, and make sure the indentation and spacing reflects the logical structure of your program. It is also a good idea to pick meaningful names for variables; "`year_now`", "`age_now`", "`another_year`" and "`another__age`" are better names than "`y_n`", "`a_n`", "`a_y`" and "`a_a`", and much better than "`w`", "`x`", "`y`" and "`z`". Although popular, "`temp`" and "`tmp`" are particularly uninformative variable names and should be avoided. Remember that your programs might need modification by other programmers at a later date.

(BACK TO COURSE CONTENTS)

1.9 Summary

We have briefly and informally introduced a number of topics in this lecture: variables and types, input and output, assignment, and conditional statements ("if statements"). We will go into each of these topics more formally and in more detail later in the course. The material here is also covered in more detail in Savitch, Chapter 1, Section 2.4 and Section 2.5.

Exercises
