Fundamental C++

Cross-platform C++ programming for beginners

Chapter 3 – draft version

3. What is programming / C++?

"Correct is better than fast. Simple is better than complex. Clear is better than cute. Safe is better than insecure." – from the C++ coding standard

In section 2.5 we have explained that a program is a file that contains instructions and in section 2.3 we have explained that a computer can only store combinations of zeros and ones, that is, binary values. Hence, all instructions must also be stored in a binary format. This format is known as *machine language*. There are probably only a handful of people in the world capable of writing a non-trivial program directly in machine language. Also, it is in general not very useful as machine language can be different for different types of CPU's.

Assembly languages, also known as assembly or asm, are more readable than machine language. Instead of binary numbers, instructions are formulated as abbreviated linguistic commands, known as *mnemonics*. Also, instead of having to refer to memory addresses, assembly can give names to them. However, the same drawbacks that apply to machine language also apply to assembly languages, which are just easier to read and understand.

To be able to program at a more abstract level, that is, independent of the type of CPU, we need a higher-level language. C and C++ are examples of such languages, though they are sometimes considered low-level languages by programmers in even higher-level languages, such as Java, for allowing direct memory access. Still, all these languages are known as *programming languages*. The exact definition of what constitutes a programming language is disputable. Suffice it to say that there are many programming languages that differ widely in their use and applicability. In this tutorial, we focus solely on C++ for its combination of efficient program execution, strong support for object oriented design and cross-platform application possibilities. Note that we in no way try to provide a full overview of C++, but instead only give an introduction that allows you to use and understand the basics, such that you are able to move on to more advanced material.

History of C++

C++ is a programming language developed by Bjarne Stroustrup, while working at AT&T Bell labs from 1979 on as an enhancement of C. It was initially called 'C with Classes', but the name was changed to C++ in 1983. C is a programming language initially developed by AT&T Bell labs' Dennis Ritchie between 1969 and 1973 for use on the Unix operating system. It soon after became available for other operating systems as well and thereby gained in popularity, since this made it possible to write code that could be easily transformed into an executable program on a wide variety of machines and on different operating systems. Moreover, due to its design, C is a language that allows for very efficient program execution. This is at the same time a drawback of C, since mistakes in a program are easily made, causing unexpected and (therefore) undesirable behaviour.

The C programming language was standardised in 1989 by the American National Standards Institute (ANSI) by a committee formed in 1983. This standard was adopted by the International Organisation for Standardization (ISO) in 1990. These standards are

known as C89 and C90, respectively, but are the same with respect to content. In the late 1990s the C language was revised, leading to the C99 standard in 1999 (ISO) and 2000 (ANSI). Not all compilers support this standard, however, due to the small demand for it. Nowadays, C++ has largely replaced C, though certainly not completely.

In 1985, C++ was first released commercially and the first edition of the book 'The C++ programming language' by Bjarne Stroustrup was published. After that, C++ kept evolving and expanding up to date. The first standard for C++ was published in 1998 by a joint ANSI-ISO committee, which thereafter processed feedback on it and published a corrected version in 2003. Most, if not all, compilers try to support this standard as much as possible, but a lot of them are still struggling with some parts of it, in particular with templates.

C++ syntax

The *syntax* of a programming language is the collection of rules that must be adhered to by the code for it to be valid. C++ has a very strict syntax. If a single character is forgotten or misplaced, the code can become invalid. Moreover, difference is made between capital and small letters in all expressions, that is, C++ is *case sensitive*. On the other hand, C++ expressions are quite easy to understand, due to the use of linguistic commands – much more than assembly languages. For example, try to guess what the program in Listing 3.1 does.

```
 1   #include <iostream>
 2   #include <cstdlib>
 3   #include <ctime>
 4
 5   int main()
 6   {
 7         std::srand(std::time(0));
 8         int number = rand() % 100 + 1;
 9         std::cout << "Guess a number between 1 and 100."
->   << std::endl;
10         int guess;
11         std::cin >> guess;
12         while (guess != number)
13         {
14               if (guess < number)
15               {
16                     std::cout << "Too low. ";
17               }
18               else
19               {
20                     std::cout << "Too high. ";
21               }
22               std::cout << "Guess again." << std::endl;
23               std::cin >> guess;
24         }
25         std::cout << "Yes! That's it." << std::endl;
26         return 0;
27   }
```

Listing 3.1: Example program

Though you may not know the precise meaning of every expression in this program, it is likely that you are able to follow what is being done and even what most of the expressions more or less mean.

You may have noticed that several levels can be distinguished in the code. In particular, groups of lines are indented the same and the lines after a `{` have a higher indentation, until a `}` is reached. The code between a pair of these curly braces is known as a *block*. Note that the indentation is not mandatory. However, braces always enclose a block of code. In a block of code, there are one or more *statements*. Each statement must be ended with a semicolon (;). Again, each statement must be ended with a semicolon! Forgetting to do so is the most common error made by people that start learning C++.

There are also lines, which are not ended with a semicolon. If you look carefully, you will see that all these lines, with the exception of the first three, are followed by an opening curly brace. These special kinds of statements are introduced in chapters 6 and 7. Until then, you only need to remember that there is no semicolon after `int main()`, but all other statements must end with a semicolon.

Normally, we put each statement on a separate line, but one statement can also span across multiple lines and more than one statement can be put on a single line, as long as each one of them is ended with a semicolon.

The C++ syntax is very liberal concerning white space – that is, space, tab and end-line characters. For example, the indentation of the blocks in Listing 3.1 is not required. In fact, we could put lines 5 through 27 all on one line by removing all the white space, but this would make the code incredibly hard to read. The same holds for the indentation of the blocks. You can avoid such mistakes as forgetting a closing curly brace by simply using a good layout for your code. Moreover, this makes the code much easier to read for others and for yourself later on.

It is considered good programming practice to make use of white space to make your code more readable – again, both for others and for yourself. Perhaps in the beginning, you will not go through

```
int number = rand() % 100 + 1;
```

any faster than through

```
int number=rand()%100+1;,
```

but as you learn C++, you will learn to process such statements faster and faster and at some point you will notice that the first instance is much easier to read than the second. Hence, it is a good idea to train yourself to write well readable code now.

Exercise 3.1 To train yourself in reading code meticulously, try to find the syntactical differences between the following pairs of statements – hence, excluding white-space. Also, name the syntax errors that you are able to spot.

a)	`if (a == b) {`
	`  x = (a + b) / 2;`
	`  y = x – b / 2;`
	`}`
		`if(a=b)`
		`{  x = a+b / 2;`
		`  y=x - b/2`
		`}`

b)	`int n = 24 * 60 * 60;`	`int n=24*60 *60`
c)	`float f = 8 * 40 / 5;`	`vloat v = 8*40 / 5;`
d)	`int n = rand() % 100+1;`	`intn=rand()%100 + 1;`

e)	`if (guess < number) {`
	`std::cout << "Too low. ";`
	`else`
	`std::cout << "Too high. ";`
	`}`
		`if (guess << number)`
		`{`
		`  std:cout <<`
		`  "Too low. "`
		`}`
		`else`
		`{`
		`  std:cout <<`
		`  "Too high. "`
		`}`

f)	`std:srand(`
	`  std::time( 0 )`
	`);`
		`std::srand(std::time(0);`

3.2. From code to program

What you have seen in listing 3.1 is code. Because we are able to create a program from this code, we have (misleadingly) designated the collection of code in the listing as a program. However, in light of the terminology used in paragraph 2.5, we cannot call a collection of characters a program, unless a computer is able to execute the file in which it is stored, which it is not. Hence, the collection of characters has to be transformed into a file containing instructions that the computer can execute. We will describe the steps in that process globally in this section.

Code is a collection of characters that adheres to the syntactical rules of a certain programming language. Therefore, code can be read and entered in any standard text editor. The code that is used to make a certain program is known as the *source code* or briefly source of that program.

The characters that constitute a piece of code have to be interpreted by the computer and (eventually) converted into instructions that can be executed by the computer, that is, machine language. The process of reading, checking and interpreting the code is termed *compiling* and the computer program that takes care of it is known as a compiler. One of the jobs of a compiler is to check the code against the syntax of a programming language. Hence, a C++ compiler will check if the code adheres to the C++ syntax. If the compiler finds any syntax errors – a part of the code not adhering to the syntax – it will display a message stating the type of error, such that the programmer is able to fix it. Learning to interpret these messages and solving the errors is an important part of becoming a programmer.

Once the code has been compiled, the separate pieces need to be put together to *build* a complete program. When all the source code of a program is spread out over several files, the process of making the connections between the different compiled versions of the different source files is called *linking*. This has to be performed before the final program can be build. In the beginning, the programs that you create will have all their source code contained in a single file. When you have learned how to (re)use multiple files, you may come across *linking errors*. These are errors that are not caused by syntactical errors,

but due to the fact that the code in one file needs to use the code of another file, but cannot find it. We will explain more about this when appropriate.

When a program has been build, we are able to *execute* it, meaning we start the program. When you create a program, you should always make sure it does what it is supposed to do. Even if you have solved all syntactical errors and, hence, are able to build your program, you cannot be certain it does what you would expect. In general we can distinct three categories of things that can go wrong:

- *Compile-time errors (compiler errors)*
  These are syntactical errors that are indicated by the compiler when you compile your program. You will not be able to build (or execute) your program until you have resolved all compiler errors. Linking errors are also compile-time errors, but they are distinct from compiler errors, as they are not caused by syntactical errors.
- *Run-time errors*
  These are errors that are caused by code that is syntactically correct, but cannot be executed by your computer, for example dividing by 0. Run-time errors will cause your program to *crash*, meaning the program execution is terminated abruptly.
- *Logical errors*
  These are errors that are not caused by a syntactical error and will not cause the program to crash, but will cause it to produce different results as expected. A well-known logical error is to use integer division, instead of floating point division, causing the result to be cut off to an integer.

We have stated that code can be entered with any conventional text editor. However, it is not very convenient to write your code this way. Also, to compile your code you will need to call the compiler program, to build your program you will need to call a builder program and to execute your program you will need to call it. To ease the amount of work required for all these actions and, hence, allow you to program much faster, people have invented Integrated Development Environments (IDE). An IDE is a program that allows you to perform all programming activities using one and the same interface. Moreover, most IDEs allow you to *debug* your program, that is, execute the statements of the program one at a time, while allowing you to see the code and the value of the variables in the code. Debugging is a very useful process to resolve run-time and logical errors.

Since there are several IDE packages available on the market, which all work in a different way, we will not describe the functioning of each one. Please consult the documentation of your IDE or the internet to find how to use your IDE. In appendix TODO you will find a brief description of how to use the Microsoft Visual Studio 2005 IDE.

3.3. Your first program

"The computing field is always in need of new clichés." – Alan Perlis

One of those clichés is that the first program you make prints 'Hello world!' to the screen, so we will stick with this tradition. Also, it is an easy way to introduce some of the basics of a console program. Create a new console application and create in it the file 'hello.cpp'. Copy the following code to the file, without the line numbers; these are just there to make it easier to explain the code.

```
1   #include <iostream>
2
3   int main()
4   {
5         std::cout << "Hello world!" << std::endl;
6         return 0;
7   }
```

Listing 3.2: Hello world program

Compile, build and run the program. You will see a command prompt window, displaying the text "Hello world!". If you only see a screen popping up and then disappearing immediately, take a look at appendix 11.1. Let us go through the lines of the code one by one to explain what everything does.

The first line is called an include directive, which is one of the compiler directives that all start with a # (pronounce: sharp). It tells the compiler that it should look in a specific file, in this case `iostream`, for commands that are used in our program. In particular, `std::cout` and `std::endl` are declared in the file `iostream`. If you right-click on an include file in your code window and click 'Open document' you can see the code in the include file.

Line 3 starts the main function. A C++ program must always have a main function and that function must always return an integer value, hence, the expression `int` before `main`. The parentheses after main indicate that it is a function. After the closing parenthesis, we start the block of code that is known as the body of a function, in this case the main function, by the opening curly bracket on line 4. The corresponding closing curly bracket is placed on line 7. Note that every opening bracket must have a closing bracket and vice versa. The opening bracket could have been placed on the previous line, but it is a good practice to give the same indentation to corresponding opening and closing brackets and the code block between them a greater indentation. This way it is easy to see in which block of code a statement is placed. Also, it makes it easier to identify missing brackets, which will cause your program not to compile. Note that some code editors perform this indentation style automatically as you type.

As stated, lines 5 and 6 make up the body of the main function. In line 5, first, the text 'Hello world!' is sent to the standard output (`cout`), followed by an end-line character (`endl`) and the whole is printed on the screen by the standard output. The statements `cout` and `endl` have to be prefixed with `std::`, because they are declared in the std namespace – std is an abbreviation for standard. What a namespace is exactly will be explained later. For now it suffices to know that this is done to avoid ambiguous names – it is perfectly

okay for the compiler if there is another cout in another namespace or in the global namespace, which requires no prefix – and that we need to type `std::` before these standard statements.

In line 6, we tell the program to exit the main function – and thereby exit the program – and return the value 0. Since the main function has to return an integer, we must specify which value is to be returned. We could as well have returned the value 1. The return value of the main function can be used by another program that launched the program we have just written or in a batch file from where our program is launched. We will not go into this, since it is not very useful in the context of this tutorial.

Though the code in Listing 3.2 is complete, something is still missing, namely a brief explanation of what is being done, that is, what the code means. For this example, the code speaks for itself mostly – if you know C++, at least – but in larger or more complex programs it can be very helpful to briefly explain what a certain part of the code does or what certain values represent. This can be done with *comments* in the code, which come in two varieties. First off, there is the line comment. If we type `//` anywhere in the code, those two characters and the rest of the line after them will be considered a comment. Secondly, there is the block comment. If we type `/*` somewhere and `*/` somewhere after it – possibly on another line – everything between and including those characters will be considered a comment. Comments are not code and are therefore not processed by the compiler, they are just skipped.

It is very useful to add comments to your code to explain what you are doing – or trying to do – both for yourself and for others. It makes it easier for someone else to understand your code and for yourself later on, when you do not remember every little detail anymore. Also, it is good practice to begin each file with a comment explaining who wrote it, when it was written and what is done by the code in that file, especially when you are doing exercises. Listing 3.3 displays the same code as before, but now with the addition of some comments.

```
 1   /* hello.cpp
 2   Author: Jelle Hurkens
 3   Date: 9 Jan 2006
 4        A program that prints the string 'Hello world!'
 5        to the screen.
 6   */
 7   #include <iostream> // cout, endl
 8
 9   int main()
10   {
11        // print the text on the screen
12        std::cout << "Hello world!" << std::endl;
13        return 0; // normal program termination
14   }
```

Listing 3.3: Hello world program with comments

At the top, we display some useful information about the file and its contents. We use a multi-line comment for clarity and ease of use. Behind the include directive in line 7, we use a line-comment to state the expressions from the included file that we have used. It can therefore quickly tell us if the inclusion of a file is still necessary after we would have

made some changes to the code. The comment on line 11 explains what we want to do in general, code-unrelated terms. The comment at the end of line 13 explains the meaning of the value 0 that is used there. Returning 0 from the main function is commonly used as an indication that the program terminated as expected. If some error occurred, another value should be returned indicating the type of error that occurred.

Of course, you most probably do not want to learn C++ just to be able to print some text to the screen. One of the reasons to use any programming language is the ability to calculate. Arithmetic in C++ can be done in a way that is very analogous to the way in which you would normally write it down. For example, the normal precedence rules apply. Let us take a look at a small example. Test this code on your own compiler to see what the results are.

```
 1   /* calc.cpp
 2   Author: Jelle Hurkens
 3   Date: 23 Jan 2006
 4        A program that calculates some numbers and
 5        displays the results on the screen.
 6   */
 7   #include <iostream> // cout, endl
 8
 9   int main()
10   {
11        // print result of calculations on screen
12        std::cout << "2 * 3 + 4 = " << 2 * 3 + 4 <<
 ->  std::endl;
13        std::cout << "2 * (3 + 4) = " << 2 * (3 + 4) <<
 ->  std::endl;
14        std::cout << "3 + 4 / 2 = " << 3 + 4 / 2 <<
 ->  std::endl;
15        std::cout << "2 + 3 / 4 = " << 2 + 3 / 4 <<
 ->  std::endl;
16        std::cout << "5! = " << 5 * 4 * 3 * 2 * 1 <<
 ->  std::endl;
17        std::cout << "A week has " << 7 * 24 * 60 * 60
 ->  << " seconds." << std::endl;
18        return 0; // normal program termination
19   }
```

Listing 3.4: Some calculations

First of all, notice the difference between the text between double quotes, such as `"2 * 3 + 4 = "`, and the numbers that are not between double quotes, such as `2 * 3 + 4`. The former is considered to be mere text by the compiler and therefore no calculations can be performed on it or within it. In fact, everything between double quotes is considered just text and when we print it to the screen we will see the exact characters that we see in the code, with the exception of escape sequences, discussed in section TODO. The latter expression is processed by the compiler and only the result will be printed on the screen.

Note that the result of the calculation in line 15 may seem a bit peculiar. We would expect a result of 2.75, but get a result of 2. This is due to the fact that the numbers in the calculation are considered integers and, hence, integer division is performed. In integer calculation, the result of every calculation is rounded down to the nearest integer. If we

would like the compiler to perform decimal division, we would have to make clear that either the numerator or the denominator is a decimal number – or both are. We can do this by adding a dot or .0 to the number, depending on the compiler. So `2 + 3.0 / 4` will give a result of 2.75, which may contain some trailing zeros when printed to the screen. Note that only one of the numbers in the division has to be decimal, since decimal calculation has precedence over integer calculation. This also holds for addition, multiplication, etc. However, if we would have written `2.0 + 3 / 4`, the result would still be 2.0, since integer calculation is performed on the division, which contains only integer numbers.

We have stated that the calculations are performed by the compiler, that is, not by the program that we have created. This shows that this program is somewhat of a special case, since normally calculations are performed as instructed by the program. The exceptionality lies in the fact that all the numbers in the calculations are *constant expressions*. A constant expression is an expression of which the value is known at compile time and whose value cannot change at run time. Hence, not only all the numbers in the program in Listing 3.4 are constant expressions, but also the text that we print on the screen. In chapter 4, we will see that there are more elegant ways in which constant expressions can be used than in this example by means of constant variables.

Exercise 3.2 Many computer programs have been written over the years that are able to play chess. Think of a way in which you could display a chessboard with its pieces in a certain state of a chess game. Write a program that uses this way to display the start position of the pieces on the board in a chess game.

Exercise 3.3 What would be the result of the following calculations when you would perform them in your code? First, find the result manually, then check your answers by writing a small program that prints the results on the screen. Remember to provide your code with useful comments.

a) 2 + 3 / 2
b) 2 - 9 / 8
c) 2 + (-9) / 8
d) 2 + 5 / 0.5
e) 3 - 1.5
f) 3 - 1.5 * 2
g) (3 – 1.5) * 2
h) 1.0 * 3 / 2

Exercise 3.4 In normal calculus, what is the value of 1 / 0? What do you think the result would be when we let the compiler or program calculate 1 / 0 or 1.0 / 0.0? Check what happens by writing a small program.

Chapter summary

- The binary format of instructions contained in a program is known as machine language. Assembly languages represent the binary values with linguistic commands. Machine language and assembly are low-level languages, as opposed to the C and C++ programming languages.
- C++ was developed as an enhancement of C. A strength of both programming languages is that programs can easily be ported to different platforms and are executed very efficiently. C and C++ have been standardised by ANSI and ISO.
- The syntax of a programming language is the collection of rules that must be adhered to by the code for it to be valid. The C++ syntax is case sensitive. A collection of statements placed between curly braces is known as a block of code. Each statement must be ended with a semicolon. The C++ syntax is very liberal concerning white-space. You should take advantage of this to make your code more readable.
- Code is a collection of characters that adheres to the syntactical rules of a certain programming language. This code has to be compiled in order to be turned into a program. The different pieces of a program need to be put together to build the final program. After a program has been build it can be executed. A program cannot be build if it contains compile-time errors. Run-time errors can occur when a program is being executed and cause the program to crash. Logical errors will cause the output of a program to be different from what was intended.
- You have made your first program. We have seen the necessary expressions to make a program; their exact meaning will be explained later. It is required to precede standard statements, such as `cout` or `endl`, with `std::`. Comments can give useful information about code. We can use line comments or block comments. Comments are not code.
- Everything between double quotes is considered to be text. Basic arithmetic calculations can be performed on numbers and the normal precedence relations hold. In integer calculation, the result of every calculation is rounded down to the nearest integer. If one of the numbers of an elementary calculation is a decimal number, the result will also be a decimal number.