Chapter 2: First Steps

Notes

- A lot of symbol characters, such as the asterisk, are used multiple times in the language to represent different operators. The compiler knows which operator is meant only by the context in which it is used.
- Operators are sometimes referred to by the symbol characters they use. For example the 'plus' operator or the 'comma' operator. This has the same affect as referring to the operator, but it is less specific and the context must be implicit.
- Operators have *associativity* which is the direction in which operators of the same precedence are evaluated. Most mathematical operators are done left to right: (x++) + (y++) will evaluate (x++) before (y++). Many unary operators are done right to left. This is to say that if you had two unary operations back to back, the right-most would be done first.
- Two's compliment system (negative integers)

Breaking into the Circle

The first steps may be you're most difficult. In learning a programming language you have to "break into the circle" as I call it. A lot of the instructions and concepts use other things which use other things which lead back to the beginning. It's a giant circle of information. It's true that the introductory circle is much smaller than the language itself, but it still is a hurdle to be overcome. I've tried to write this to break into it as easily as possible.

The 'Hello World' Program

In writing a book on programming it's expected that the first program explained will be the legendary 'Hello World' program. This program simply has to output the text 'Hello World' to the screen and exit. The C++ source code to do this is below[1]:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      cout << "Hello World" << endl;
06      return 0;
07  }
```

[1] You may see programs using '`void main()`' or just '`main()`'. These are valid as well and will be explored in Chapter 7: Functions.

In all of the source code I will present to you, a column of numbers will be to its left. These numbers represent the lines on which instructions occur. It will make it easier for me to show you which lines I am describing. But these are simply a visual benefit, do not type them in as part of your source code or it will not compile. This source should simply be typed as:

```
#include <iostream.h>

main()
{
    cout << "Hello World" << endl;
    return 0;
}
```

Now type the source code to this program into your text editor and save it as '`hello.cpp`'. Now compile that into the executable file '`hello.exe`'. Don't worry; you can use those same filenames on whatever platform you're using. Some platforms (like Linux or Unices) do not require a special extension to mark a file as executable; my advice, however, is to do it anyway so you can tell what each file is immediately by its name. Lastly run the file from a command console. You should see the output:

```
Hello World
```

On many systems, DOS/Windows for example, there will be an additional blank line after the text 'Hello World':

```
>hello.exe
Hello World

>_
```

On other systems there will be none. This is just how these systems deal with programs that output console text. Some will automatically add an extra new line in case the program doesn't, while others will not. But I digress; let us break down this program line by line.

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

This first line does quite a few things. But all you need to know at this point is that it *includes* the file '`iostream.h`' which is like a source file, but is called a *header file*. Basically this gives the program the functionality to do some basic input and output. In our case, it gives the program the ability to output text, specifically 'Hello World'.

```
02
```

This is a blank line. Since there is nothing here, it is skipped. There will be a quiz on what a compiler does with a blank line later on so remember this. ☺ I hope for your sake that you realize I'm joking.

```
03  int main()
```

Here a *function* called '`main()`' is defined. All C++ programs must contain this to be compiled. A function is a group of C++ instructions that perform some sort of task. Our task in this case was to output the text 'Hello World'. The program knows to execute this function and it's task because of it's special name: '`main()`'. Quite simply, this is where the program begins. Chapter 7 is all about functions.

```
04  {
```

The opening curly brace signifies the beginning of the function we just defined called '`main()`'. All instructions to be run when the program starts will be inside the pair of curly braces associated with '`main()`'.

```
05      cout << "Hello World" << endl;
```

Ah, the most interesting line. It is instructions to output the text 'Hello World' to the console you have opened. The output of characters to the command console is also known as *printing* as in printing to the screen and not a printer. Now, '`cout`' is actually an *object* and the less-than sign pairs are *operators* that are part of '`cout`'. However, I will not get into these sorts of things until much later. You will have to be content to simply *use* the mysterious '`cout`' and '`<<`' to output things to the screen.

```
06      return 0;
```

This sets the exit code of our program to zero and ends our program. It should always be the last thing before the closing curly brace:

```
07  }
```

Here the function known as '`main()`' ends as well as our program. After all C++ instructions up to this curly brace have been processed, the program will end and control will return back to you at the command console.

Let's get back to the output. You can plainly see 'Hello World', but what of the strange '`endl`' following it and what of all these '`<<`'? First off, 'endl' is a special symbol that means 'end of line' or 'go to the next line here'. For example, try changing line 5 to the following:

```
05      cout << "Hello" << endl << "World" << endl;
```

After compiling and running this you will see the output:

```
Hello
World
```

The words are now on separate lines. This is because of the 'endl' we put in between them. Notice for each new thing that you want to output, you need to put a '<<' in front of it. When you want to output text, as we have done, you must surround it with double quotes. The double quotes will not be printed along with the text; they simply tell exactly what you want to output.

All of the lines, as you've no doubt noticed, are executed in precise order starting from the top-most. The order in which instructions (as the line they are on does not matter) are executed in C++ always follows this pattern.

Syntax

Now it's time to get technical and explain some things that way. Syntax is the rules of a language. We have syntax in English too, but we call it by a different name. However, syntax in programming must be followed exactly unless we want to torture ourselves and/or our compiler. You cannot escape the rules of a programming language, like you can with poetry in English. You can be a poetic programmer and still follow the rules. For there are exceptions to the rules, but they simply end in more rules. ☺

Most of what a C++ program is made of is *statements*. A statement is a set of instructions that usually ends with a semi-colon. The exceptions to the 'ending with a semi-colon rule' are control statements which are special language constructs, usually for flow control. Almost all statements, however, *will* end in a semi-colon.

What the heck is a statement and why do we have them? A statement is like a sentence in English. And this analogy works out pretty well. In English you have to end all sentences in a period except, for example, titles. This is much the same in C++ source code. The function definition '`main()`' is just like a title. It says "read me first". Then the statement on line 5 is just like a sentence.

Since statements represent something being done, i.e. instructions, the term for when their action(s) is performed is known as executing. This isn't the killing kind either. When a statement is "executed" it is performed as per the instructions it contains.

If you know a bit of English, you know that the main points of every sentence are its nouns and verbs. The nouns are the 'things' and the verbs are what the things are doing or what's being done to them; i.e. the 'action'. Again, it is much the same in C++, except the actions are known as *expressions* and the participants in this are the *operators* (like verbs) and *operands* (like nouns). I just did overkill on new glossary terms, eh? ☺

Smooth Operator

An operator is like a verb in English. It describes the action or *operation* to be done. The participants in this action are known as the operands. Now there are three types of operators I'll cover here: *nullary*, *unary*, and *binary*. These simply describe the number

of operands used with the operator. The first, nullary, is an operator that has no operands. Next is unary which is an operator that accepts only one operand. Lastly is binary. Now, you already know that the binary numbering system is base two, so it should come as little surprise that a binary operator takes two operands. Although there *are* operators which take more than two operands, they are not common *and* they break down into multiple, normal operations anyway.

The operand of a unary operator typically appears to the left, but not always. In the case of two operands, with a binary operator, they appear on either side. There is just no place for more than two operands!

Nullary:	<operator>
Unary:	<left operand><operator> *or* <operator><right operand>
Binary:	<left operand><operator><right operand>

All operations boil down to having one, two, or no operands. If you have had any sort of mathematics background, it is the same there. Computers are very mathematical as I have explained before and the relation here should come as no surprise. In math every calculation eventually breaks down to a series of calculations between two numbers. The calculation being done, whether it is addition, subtraction, or another, is the operator. The numbers involved in the calculation are the operands. For example:

4 + 5

The '+' is a binary operator whose operation is to add its left operand to its right. The result is 9. If this were part of a complex formula the result (9) would become an operand itself:

4 + 5 + 8

The '4 + 5' is done first, then the result of that (9) is added to '8'. Even though there are more than two numbers, the individual operations to add them all together only take two. Any amount of operators, even one, with their operands is known as an *expression*. Multiple operations combined are a *complex* expression, but it is still an expression. You won't usually hear the term 'operation' at all, but you will most definitely hear 'expression'. This is fine. The terms 'expression' and 'operation' are practically synonymous. From now on I will use the term 'expression'; only in explicit situations will I describe something as an 'operation' (usually when it involves a single operator). The above (4 + 5 + 8) is a complex expression. Line 5 of our 'Hello World' program is a complex expression. Can you identify the operators and operands in it?

```
05      cout << "Hello World" << endl;
```

Both of the '<<' are operators of course. The rest, save the semi-colon, are operands: '`cout`', '`"Hello World"`', and '`endl`' are operands.

The '<<' is the 'insertion operator' and it performs an 'insertion operation'. In this operation the right operand is *inserted into* the left operand. The result of an 'insertion' operation is always the left operand. So, line 5 breaks down into the following operations:

```
cout << "Hello World"
cout << endl
```

These operations are performed one at a time. First '`"Hello World"`' is *inserted* into '`cout`' and next '`endl`' is inserted into '`cout`'. So as not to confuse you too much, I'll let on a little about '`cout`'. It is an abbreviated name for 'console output'. By inserting things into it they are pushed into the programs standard output stream, which of course flows onto your monitor in the command console. You can insert practically anything into '`cout`' and it will translate it into something that can be shown in the command console as text. For example, using '`endl`' doesn't put the text "endl" on the screen, but moves the output cursor down a line. Isn't it nice for '`cout`' to do all of that for you?

All the operations we have dealt with up to this point represent a *result*. All binary and unary operators will yield this. The result is the yield of the operation; i.e. what comes out of it. With the insertion operator the result is always the left operand. Therefore after the operation is performed, the left operand is again used in the next operation.

My Dear Aunt Sally

A complex expression is more than one operation. In the 'Hello World' program the operations were evaluated from left to right. First '`"Hello World"`' and then '`endl`'. This is not, however, always the case. Remember your math:

```
5 + 6 * 7
```

What is done first in this? First '6' is multiplied by '7'[2], next the result of that is added to '5'. Just as in math, certain operators have higher precedence than others; hence the term *operator precedence*. The higher the precedence of an operator, the earlier it is performed. But just as in math, isn't there something you can do about it? Yes, you can use parenthesis to make a particular expression or operation have the highest precedence. If we wanted to add '5' to '6' and then multiply *that* result by '7', we'd write it:

```
(5 + 6) * 7
```

Unfortunately, the line in '`hello.cpp`' that outputted 'Hello World' cannot be modified to demonstrate operator precedence. For that we'll have to use some very math-like expressions.

[2] In programming the asterisk is for multiplication, not a dot or an 'x'.

Outputting Numbers

A number is dealt with differently than text. Text must always be surrounded by double quotes, numbers must not. This is because the compiler can understand a number, but text could be relatively *anything*. The compiler knows that a number only contains specific digits. For example, to output the number '5' we could write the following:

```
cout << 5;
```

This is the decimal number 5 because there are no special distinguishing marks. By default '`cout`' will output a number in its decimal form. Later on we'll look at how to make it output numbers in hexadecimal or even binary. To put a number in your code it must be formatted very specifically. Decimal numbers have no extra markings. Hexadecimal numbers must be preceded by '0x', and octal numbers (god forbid) must be preceded by a '0'. There is no way to represent a binary number directly in C++; you'll need to use hex for this purpose. But *remember*, '`cout`' will display the number in its *decimal* form by default. Take the following lines:

```
cout << 0xF << endl;
cout << 15 << endl;
cout << 017 << endl;
```

If you were to compile and run a program with these three lines, and I encourage you to try this, you will see the same number on three lines. Hexadecimal '0xF' is '15' in decimal and octal '017' is also '15' in decimal. I can't remind you enough that the *value* of the number does not change. We're still talking about fifteen (☺☺☺☺☺☺☺☺☺☺☺☺☺☺☺), just *representing* it differently.

Not only can we output just *numbers*, but mathematical expressions. Let's output the result of '(5 + 6) * 7':

```
cout << (5 + 6) * 7 << endl;
```

The output of this will, of course, be '77' on its own line. However, what's more important is this is a great example of operator precedence. This expression breaks down like so:

```
cout << (5 + 6) * 7 << endl;
cout << 11 * 7 << endl;
cout << 77 << endl;
cout << endl;
```

Each of the individual operations has been outlined so you can see them clearer. Notice that the insertion operator has the *lowest* precedence of all the others. The parenthesis cause '5 + 6' to be done first, then the multiplication operator (*) is used with '11' and '7', next the result (77) is outputted, and lastly a new-line follows.

What would make this better is if we include some text. Below is a program that calculates '155/3':

```
01  #include <iostream.h>
02
03  int main()
04  {
05      cout << "155 / 3 = " << 155/3 << endl;
06      return 0;
07  }
```

If you compile and run this program, you'll notice that the result is rounded off. There is no decimal point. The result should be '51.666666…', but instead it's only '51'. The reason for this is *integers*. In programming there are two types of numbers the computer uses natively: integers and floating-point.

Integers

An integer number, quite simply, is a whole number. Deep down, in the bowels of the computer, an integer is stored using a plain sequence of bits that represent its value. One thing you might have noticed about our numbering systems is that we always dealt with *whole* and *positive* numbers. In C++ the decimal numbering system is the only one used to show negative or fractional numbers. But then how does a computer store negative numbers if it uses binary? The answer is a *sign bit*.

An integer that uses a sign bit is known as a *signed integer*. By default on most systems, integers are signed. In a signed integer the system will reserve one of its bits to use as a sign. How the computer uses this sign is pretty dependent on the CPU. It may decide to use '0' for negative and '1' for positive or vice versa. In my experience, however, it is usually '1' for negative and '0' for positive.

Getting back to the whole-number side of integers, this cannot be avoided. Any mathematical operation you do with integers will result in a whole number. Fractional values are simply dropped at *each* operation, not at the end of a complex expression. But really, dividing is the only place where this becomes a problem. And there is a useful operator especially for integers: *modulus* or '%'. This performs a 'modulo' operation which, strangely enough, results in the *remainder* of an integer division. For example you can fit four into ten, two times, but there is a remainder of two:

10 / 4 = ☺☺☺☺ ☺☺☺☺ ☺☺

A divide operation would return two because that's how many times four goes into ten. A modulo operation would return two as well, because that's what's left over after four goes into ten twice. If you don't believe me, put these two lines in a program and run it:

```
cout << "10 / 4 = " << 10/4 << endl;
cout << "10 % 4 = " << 10%4 << endl;
```

Let's try a more complicated example, one where the result isn't the same for both operations. What is fifteen divided by seven? Well, seven can go into fifteen twice, but there is a remainder of one. Thus the division result would be two and the modulus result would be one.

Floating-Point

Yes, it's *floating* point. The point in question is the decimal point used in representing fractional numbers. The reason for the name is because of how they are stored. A floating-point number isn't simply a bunch of bits that equate to its decimal value. It is a bunch of bits whose values are used in conjunction with a complex algorithm to represent a two part number: whole and fractional. Floating-point numbers are very strange and so we won't have much dealing with them.

For a number to be floating-point you simply have to give it a fractional part, even if that's just zero. For example, if we wanted to make '155 / 3' accurate we would use '155.0 / 3.0':

```
05      cout << "155 / 3 = " << 155.0/3.0 << endl;
```

This will output something like:

```
155 / 3 = 51.6667
```

Since this would result in an infinite number of sixes, it chops it off at some point and makes the last six a seven.

Limits

There are limits to the size of these numbers in your program. Recall in a previous chapter, I spoke of storage units. A CPU can only natively use numbers that are so big. Every number must occupy storage somewhere in the computer. And it will use one of the unit sizes that I detailed. For example, if you tried the following:

```
cout << 1234567890123456789 << endl;
```

You would get a strange error indeed. The number is bigger than anything the compiler expects so you may get a strange error. You'd hope to get something like, "*Number 'n' exceeds limits*", but for example in Microsoft Visual C++ you get "*'operator <<' is ambiguous*".

Once you get into choosing where the numbers are stored (variables) I will show you ways of discovering the largest possible *native* integer value. Now, floating-point value limits are different than integers because of the way they are stored and represented. You

can enter a floating-point number with practically any number of fractional digits and the compiler will accept it:

```
cout << 1.234567890123456789 << endl;
```

Again, this is because of how they are stored. The value stored will not be the exact one that you entered. The value output will be truncated as well. However, output truncation using '`cout`' is partially due to '`cout`' itself and we will look at how to get around this later. A floating point numbers have a property known as *precision*. The precision of a floating-point number can very, but it is expressed in digits and said as 'points'. For example a floating-point that is precise to seven digits is known as a seven-point precision floating-point number. This means that up to seven numbers past the decimal point will be valid, those following the seventh digit (if there are any) are garbage numbers that are practically there just to fill space. So, taking our above number with seven-point precision, it would only store '`1.2345678`' validly; everything past that would be garbage data.

White Space

In between statements, their expressions, their operators, and their operands is *white space*. This is a term for character data that is skipped. In C++ white space consists of new lines, tabs, and spaces. What this means is it treats all of these things as one contiguous thing: white space. With this knowledge you could mangle our first program and it would still compile and run fine:

```
#include <iostream.h>
                int
main
(
) { cout
<< "Hello World"
                          <<
endl ; return 0
;}
```

It's a strange way to see the program, but it still works doesn't it? This gives you control over how your program is visually structured. There are whole volumes dedicated to code writing etiquette and I will no doubt make one myself. ☺

Please note that I did not modify any lines beginning with a pound sign (#); there was only one besides. I could, but I won't. Any line that begins with a pound sign is not necessary C++ source code and is special. For this reason they will always appear drab and unchanged until you learn about pre-processing.

Comments

Along with spacing your code in a way that's readable, you can also add text to your source file that is not compiled. This means it is skipped in the same way white space is. There are two comments you can use in C++: *single-line* and *block*.

A single-line comment begins with two forward slashes (/ /) and ends with a new line character:

```
// this is my single-line comment
```

A block comment begins with a forward slash and an asterisk and ends with the same in reverse order:

```
/* this is my block comment */
```

Since a block comment doesn't require a new line character, it can be in the *middle* of a statement, or encompass whole paragraphs of code *and* single-line comments:

```
/* this is my block comment that includes a single-line comment
// here is a single line comment
more block comment stuff */
```

I suggest not using block comments at all. There may be times that you want to temporarily erase a bunch of source by *commenting it out*. If you have lots of block comments you will have to resort to pre-processor commands, but if you have lots of single-line comments you can simply block comment all of them out. To comment something 'out' means to make a bunch of text part of comment so the compiler ignores it.

Comments are used to document the source code. They should be used to explain logic and processes (especially complex ones), but not obvious things. For example, the following comment placed above line 5 would be *completely unnecessary*:

```
// outputs 'Hello World' and a new-line character
```

We can already discern that from the statement itself. Anyone who knows the first thing about C++ (or has read this chapter☺) will know that. You have to assume a certain level of competency when you write comments in a program. You should make note of these expectations somewhere so that people without the required expertise don't get frustrated by what's written. For example, if a program is making the necessary calculations for time travel there should be a comment saying, "If you don't know physics, you need to smoke crack to understand this".

Also, in using comments effectively you should create a uniform blurb at the top of every source file you create to identify it, its purpose and content, its author, and the day it was created. You'll thank me later if you start doing this now!

Summary

In this chapter we learned how to write our first C++ program which printed the text 'Hello World'. Following this I wrote how you can output other text, numbers, and even new-line characters. Besides output, I explained the basic syntax of C++ programs followed by detailed information about the pieces that make it up. Next you were introduced to a form of storage in C++ with the two numbering types. Lastly, I showed you how a C++ source file can be written with self-documentation using white space and comments.

By this point I expect you to be able to write a C++ program that can output text, numbers, arithmetic results, and new-line characters. All of the in-depth explanations are for your benefit, but are not required to be remembered to move on. Text was the important lesson in this chapter and in the next, as we learn about variables; you will be tested further with numbers.

Test Your Knowledge

Questions

1. Who's your daddy?

Programming Exercises

1. Write a program called 'name' (`name.cpp`/`name.exe`) that outputs your full name.
2. Change '`name.cpp`' so that each of your names (first, last, middle if you have it, etc.) is outputted in a single '`cout`' operation. But make sure that your name is still output to the same line.
3. Change '`name.cpp`' so that each of your names is on a separate line in the output.
4. Write a program to clear the screen by simply outputting new lines.
5. Use white space creatively, to make '`name.cpp`' look like a pillar.
6. Using single-line comments, add a header to '`name.cpp`' that tells the name of the file, what it does, who wrote it (that's you), and the date it was created.

Knowledge Answers

Questions

1. Gotcha!

Programming Exercises

These answers might be different if you don't have the same name as me (I hope not!) and if you were not as creative. ☺ Kidding! As long as you could do the exercises, it's good! Here's what I came up with:

1.

```
01  #include <iostream.h>
02
03  int main()
04  {
05      cout << "Neil Caleb Obremski" << endl;
06      return 0;
07  }
```

2.

```
01  #include <iostream.h>
02
03  int main()
04  {
05      cout << "Neil ";
06      cout << "Caleb ";
07      cout << "Obremski";
08      cout << endl;
09      return 0;
10  }
```

3.

```
01  #include <iostream.h>
02
03  int main()
04  {
05      cout << "Neil" << endl;
06      cout << "Caleb" << endl;
07      cout << "Obremski" << endl;
08      return 0;
09  }
```

4.

This will only work if your command console has twenty-five lines or less. Make sure the one you made works for you. Yes, this is awkward, especially if you adjust the command console size (it no longer works), but we'll come back to this later on to fix it up.

```
01  #include <iostream.h>
02
03  int main()
04  {
05      cout << endl << endl << endl << endl << endl
06           << endl << endl << endl << endl << endl
07           << endl << endl << endl << endl << endl
```

```
08           << endl << endl << endl << endl << endl
09           << endl << endl << endl << endl << endl;
10      return 0;
11  }
```

5.

This was more difficult than I thought it would be! ☺

```
01  #include <iostream.h>
02
03  int     main
04  (   ) { cout
05  << "Neil" <<
06  endl; cout<<
07  "Caleb"   <<
08  endl; cout<<
09   "Obremski"
10  <<    endl ;
11  return 0 ; }
```

6.

```
01  // File ...... name.cpp
02  // Author .... Neil C. Obremski
03  // Created ... 08 OCT 2001
04  #include <iostream.h>
05
03  int     main
04  (   ) { cout
05  << "Neil" <<
06  endl; cout<<
07  "Caleb"   <<
08  endl; cout<<
09   "Obremski"
10  <<    endl ;
11  return 0 ; }
```