The Classic First Program

Here is a version of the classic first C++ program. It writes “Hello, World!” to your screen:

```
// Hello world example.
#include <iostream>

int main()
{
    std::cout << "Hello World!";
    return 0;
}
```

Consisting of just seven lines of code, this small program contains everything we need to look at the basic of a C++ program. We are going to cover each aspect of this program in more details. The aim of this program is to familiarize ourselves with some core concepts before covering them in more detail as we progress.

Before our program get much more complicated, we should see how C++ handles comments. Comments help the human readers of our program. They are typically used to summarize an algorithm, identify the purpose of a variable, or clarify an otherwise obscure segment of code. The compiler ignores comments, so they have no effect on the program behaviour or performance.

There are two kinds of comments in C++: single-line and paired. A single-line comment starts with a double slash (//) and ends with a newline. Everything to the right of the slashes on the current line is ignored by the compile. A comment of this kind can contain any text, including additional double slashes. The other kind of comments uses two delimiters (/* and */). The compiler treats everything that falls between the /* and */ as part of the comment.

Starting for the top, we have a **pre-processor directive**:

```
#include <iostream>
```

The ***include*** directive is a very common directive that you’ll see in most C++ files, and it means “**copy here**”. So, in this case, we’re going to copy the contents of iostream file into our application, and in doing so, allow ourselves to use input/output functionality it provides.

Next, we have our entry point, **main()**.

```
int main()
```

The main() function is where your C++ program will kick-off. All programs will have this function defined and it marks the start of our application-the first code that will be run. This typically your outer-most loop because as soon as the code in this function is complete, your application will close.

Next, we have an IO statement that will output some text to the console:

```
    std::cout << "Hello World!";
```

Because we have included the **iostream** header at the start of our program, we have access to various input and output functionality. In this case, std::cout allows us to send the text to the console, so when we run our program, we see that the text “Hello World!” is printed.

Finally, we have a **return** statement:

```
    return 0;
```

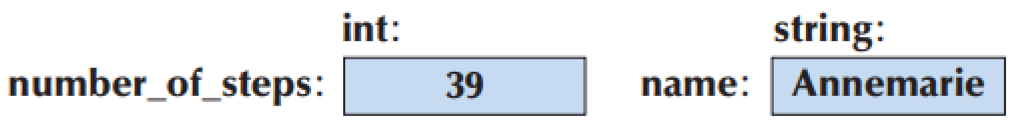
This signals that we’re done in the current function. The value that you return will depend on the function, but in this case, we return 0 to denote that the application run without error.

Keywords

Keywords are words that are reserved by C++. Thus, we cannot use them in our program for anything other than their intended purposes. For example, a common keyword is “**if**”. So you would not be able to define a variable or function of that name. it’s these keywords that structure the C++ language, and it’s through their use that we instruct our program on what it should be doing. Some of these words define basic data types, some of them are statements to define program flow, and others define object and scope.

Variables

We can do nothing of interest with a computer program without storing data in memory. The places in which we store data are called ***objects***. To access and object we need a ***name***. a named object is called a ***variable*** and has a specific ***data type***. The type of a variable determines what can be put into it and which operation can be applied to it. The data items we put into variables are called ***values***. A statement that defines a variable is called a ***definition*** and definition can provide an initial value. We can visualize variables like this:



Note that you can not put values of the wrong type into a variable.

Data Types

As we have learned, we store data in variables. For example, name, age, or the price of food items. Given these are different types of data: alphabetical, numerical, and so on, we store them in different variable types. It's these types that we're going to be taking a look at now, as it's important to use the correct variable types for the data you want to store.

Built-In Types

Now we can look at the core set of fundamental data types that C++ provides us with. These types will serve your need most of the time, and you don't need to do anything special to use them; they're part of the language. These built-In types are as follows:

- **bool**: the bool type stores either a true (non-zero) or false (0) value and has a size of one byte.
- **int**: the int type is used to store integers and is typically four bytes in size.
- **char**: the char type is used to store a single character. this data type is one byte in size.
- **float**: the float type represents single-precision floating-point numbers and is typically 4 bytes in size.
- **double**: the double type represents double-precision floating-point numbers and is typically 8 bytes in size.
- **void**: the void type is a special type that denotes an empty value. You cannot create objects of the void type. However, it can be used by pointers and functions to denote an empty value, for example, a void pointer that points to nothing, or a void function that doesn't return anything.

Type Modifiers

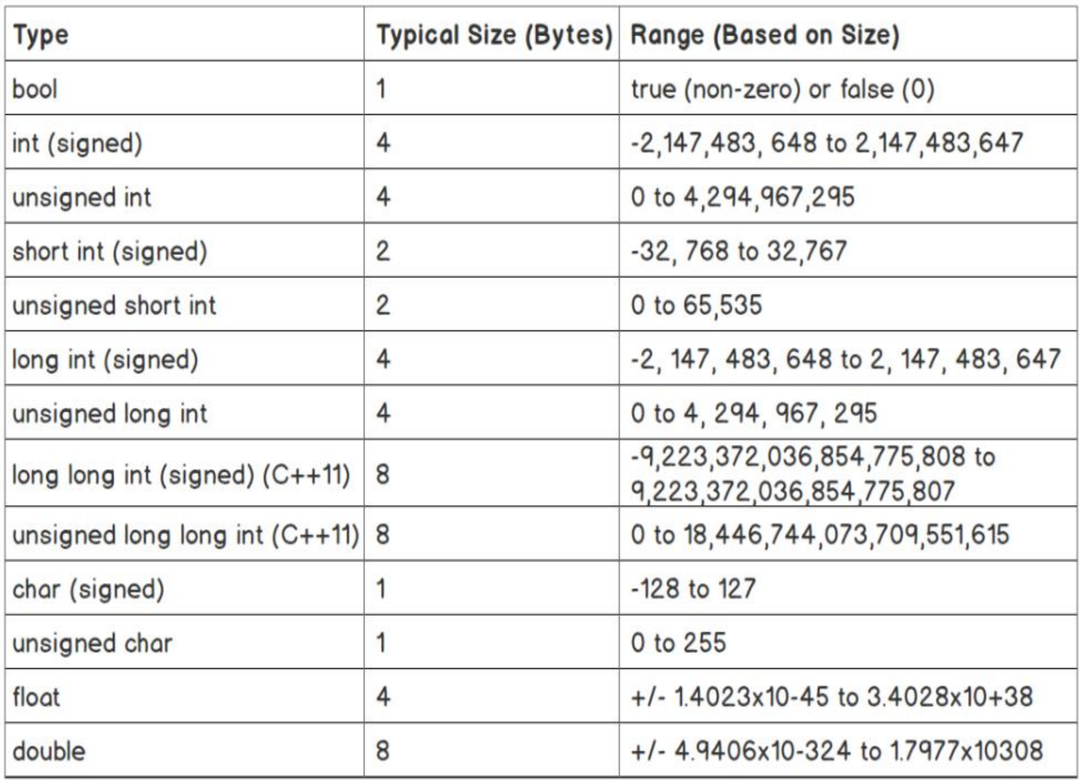
Type modifiers allow us to change the properties of the built-in data types. The following modifiers are available to us:

- **signed**: the signed keyword specifies that our variable can hold both positive and negative values.
- **unsigned**: the unsigned keyword specifies that our variable should only hold positive values.
- **long**: the long keyword ensures that our variable will be at least the size of an int, typically will be 4 bytes.
- **long long** (C++11): the long long keyword, added in C++ 11, ensures that our variable will be greater in size than long; typically, this will be 8 bytes.
- **short**: the short keyword ensure that our variable has the smallest memory footprint it can.

Note: the exact size of data types depends on factors such as the architecture that you're working with and what compiler flags are set.

Reference Table

Here is a table of the basic data types provided by C++ with a selection of type modifiers:



Type	Typical Size (Bytes)	Range (Based on Size)
bool	1	true (non-zero) or false (0)
int (signed)	4	-2,147,483, 648 to 2,147,483,647
unsigned int	4	0 to 4,294,967,295
short int (signed)	2	-32, 768 to 32,767
unsigned short int	2	0 to 65,535
long int (signed)	4	-2, 147, 483, 648 to 2, 147, 483, 647
unsigned long int	4	0 to 4, 294, 967, 295
long long int (signed) (C++11)	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long int (C++11)	8	0 to 18,446,744,073,709,551,615
char (signed)	1	-128 to 127
unsigned char	1	0 to 255
float	4	+/- 1.4023x10-45 to 3.4028x10+38
double	8	+/- 4.9406x10-324 to 1.7977x10308

Declaring Data Types

In this section we are going to declare a number of different variables, with and without type modifiers, and print out their size using the **sizeof** operator. Here are the steps to complete the exercise:

1. we'll start by defining a number of variables using three of the types form preceding table:

```
int myInt = 1;
bool myBool = false;
char myChar = 'a';
```

2. the **sizeof** operator will give us the size of our variables in bytes. For each variable defined previously, add an **output** statement that will print its size:

```
std::cout << "The size of an int is " << sizeof(myInt) << ".\n";
std::cout << "The size of a bool is " << sizeof(myBool) << ".\n";
std::cout << "The size of a char is " << sizeof(myChar) << ".\n";
```

3. the complete code looks like this:

```
#include<iostream>

using namespace std;

int main()
{
    int myInt = 1;
    bool myBool = false;
    char myChar = 'a';
    std::cout << "The size of an int is " << sizeof(myInt) << ".\n";
    std::cout << "The size of a bool is " << sizeof(myBool) << ".\n";
    std::cout << "The size of a char is " << sizeof(myChar) << ".\n";

    return 0;
}
```

Conventions for Naming Variables

variable name in C++ composed of letters, digits, and underscore character. The language imposes no limit on name length. However, it must begin with either a letter or an underscore. Variable name are case-sensitive, upper and lowercase letters are distinct. Furthermore, reserved keywords can not be used as variable name.

Operators

We learned about the various data types provided by C++, and how we can use them to store and represent the data within our systems. In this section, we will take a look at operators, the mechanisms by which we assign and manipulate this data.

Operators come in many shapes and sizes, but in general, their role is to allow us to interact with our data. Assigning a value, modifying it, or copying it, this is all done through operators. In this section, we will cover the following operators:

1. **Arithmetic operators:** arithmetic operators are those that allow us to perform mathematical operations on our data. These are very self-explanatory and straightforward to use. For example, in order to add a number, you simply use the “+” sign as you would anywhere.
   Let’s take a quick look at our four basic operators: **addition, subtraction, multiplication**, and **division**. As stated previously, these four operators have the same symbols that you’d use day to day, so they should be familiar. The following example implements all four types of arithmetic operators:

```
// Arithmetic operators.
#include <iostream>
#include <string>

int main()
{
    int addition = 3 + 4;
    int subtraction = 5 - 2;
    int division = 8 / 4;
    int multiplication = 3 * 4;

    std::cout << addition << "\n";
    std::cout << subtraction << "\n";
    std::cout << division << "\n";
    std::cout << multiplication << "\n";
}
```

We can use both variables and constants in these operations. They are interchangeable. Here is an example:

```
    int myInt = 3;
    int addition = myInt + 4;
```

In this code snippet, we add the value 4, a constant, to myInt, a variable. The outcome of this is the addition variable will now have a value of 7.
The final arithmetic operator we'll look at is the modulus operator. This operator returns the remainder of an integer division and is presented by the % symbol:

```
// Arithmetic operators.
#include <iostream>
#include <string>

int main()
{
    int modulus = 11 % 2;
    std::cout << modulus << "\n";
}
```

2. **Unary operators:** the operators that we have used had a value, typically called an operand, on either side of them: rhs and lhs. Unary operators are those operators, however, that take only one value and modify that. We will take a look at minus (-), increment (++), and decrement (--).
   Lets start with the minus (-) operators; this allows us to manipulate the sign of a value. It is fairly straightforward, when placed in front of a value it will turn a negative value positive, and positive value negative.

```
// Negation example.
#include <iostream>
#include <string>

int main()
{
    int myInt = -1;
    std::cout << -myInt * 5 << std::endl;

    myInt = 1;
    std::cout << -myInt * 5 << std::endl;
}
```

We can see from the output that the sign of the output is opposite to that of the variable since we're using it with the minus operator.
The other unary operators we are going to look at are increment (++) and decrement (--). These two operators allow us to increase or decrease a value by one, respectively. In the following code, we define a value, then increment or decrement it, and view its value:

```
// Increment/Decrement example.
#include <iostream>
#include <string>

int main()
{
    int myInt = 1;
    std::cout << ++myInt << std::endl;
    std::cout << --myInt << std::endl;
}
```

In this simple example, we have defined a value as 1, incremented it, and then immediately decremented it again, printing its value at each stage.

3. **Assignment operators**: assignment operators allow us to assign values to our objects. We have used this operator many times throughout this lecture. It is one of the most fundamental operations in programming, but always, there is more that we can learn about these operators. the most basic assignment operator is where we take a value and assign it to a variable, as follows:

```
    int myInt = 5;
```

We are familiar with this, but what we might not be familiar with is the concept of combining these with arithmetic operators. let's imagine a scenario where we need to increment a value by 5. We could do this as follows:

```
    myInt = myInt + 5;
```

We take the value of myInt, and add 5 to it, and then assign it back to the original variable. We can do this in a more refined way, however, by combining the two operators together. The assignment operator can be preceded by an arithmetic operator to achieve this, as follows:

```
    myInt += 5;
```

This is the case for any of the arithmetic operators, they can precede an assignment operator and their effects are combined. This can be seen in the following example application.

```
// Assignment Operators Example.
#include <iostream>
#include <string>

int main()
{
    int myInt = 5;

    myInt += 5;
    std::cout << myInt << std::endl;

    myInt -= 5;
    std::cout << myInt << std::endl;

    myInt *= 5;
    std::cout << myInt << std::endl;

    myInt /= 5;
    std::cout << myInt << std::endl;

    myInt %= 5;
    std::cout << myInt << std::endl;
}
```

Note: Relational and Logical operators will be covered with the control flow statements.