C++

"C makes it easy to shoot yourself in the foot;
C++ makes it harder, but when you do it, it blows your whole leg off."

~ Bjarne Stroustrup~

What C++ Really Is?

C++ is the successor to the C programming language and was invented by Bjarne Stroustrup in the early 80's.

C++ is a multi-paradigm programming language. It extends C with the object oriented concepts of classes, inheritance and dynamic (run-time) binding. C++ is a strongly typed language which means that all entities int a program must be typed. A declaration must be made to the compiler to state the kind of information which they store.

C++ standards so far:

- cpp98
- cpp03
- cpp11
- cpp14
- cpp17

The number indicates the year when the sandard was release.

The question is: why would I want to learn C++? So, C++ is pretty much the most used language when you need to write fast codes that performs well or if you're writing for a weird architecture or platform and you need to the code run natively. If you want to control the hardware C++ is for you. Game industry uses C++ as well, for example game engines as Unity, Unreal or Frostbite are all written in C++. The biggest reason to use C++ is the direct control over the hardware.

How C++ works?

You write your code in C++ and then you pass your code into a compiler and that compiler will output machine code for your target platform. Machine code is the actual instructions that youre device's CPU will actually perform. So using C++ we can literaly control every instruction that the CPU executes. C++ runs almost on every platform (Windows, macOS, Unix, iOS, Android, XBox, PS, Nintendo), you just need a compiler that would output machine code for that platform.

Note: just because your code is native doesn't mean it it's going to be fast. If you write bad code in C++ it's gonna be slow.

The basic workflow of writing a C++ program is you have a serious source files which you write actual text in and then you pass it to the compiler which compiles it into some kind of binary. Now that binary can be some sort of library or it can be an actual executable program.

Let's take a look at a very basic C++ program

```
#include <iostream>

int main()
{
    std::cout << "Hello World" << std::endl;
}
```

First of all the `#include <iostream>` statement.

Now this is something called a **preprocessor statement**, anything that begins with a hash (**#**) is a preprocessor statement. The first thing that a compiler does when it receives the source file is it preprocesses all of your preprocessor statement. That's why they called preprocessor statements, because they happen just before just before the actual compilation. What `include` will do is find a file and take all of the contents of that file and just paste it into this current file. These files that you include are typically called **header files**. The reason to include the `iostream` is because we need a declaration for function called `cout` , which let us print stuff to our console.

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

It is very important. The main() function is called the **entry** point. It is the entry point for our application. That means when we run our application our computer starts executing code that begins in this function. As the program is running the computer will execute the lines of code we type in order.

Those who are familiar with functions you might notice that the return type of `main()` is actually an int, however we not returning an integer. That's because the `main()` function is actually a special case, you don't have to return any kind of value from the `main()` function. If you don't return anything it will assume that you returning zero this only applies to the `main()` function.

Lets take a look at the next line -

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

These left angular brackets (**<<**) which look kinda bit shift left operator are actually just an overloaded operator. So you need to think of them as a function. In C++ operators are just functions. So what we actually doing here that is pushing this **"Hello World"** string into this cout and then we pushing the endl. It

tells our console to advance to the next line.

How do we get from this text file an executable binary file? Basically we go trough a few stages.

First the `#include <iostream>` once our preprocessor statements have been evaluated our file gets compiled. This is the stage where the compiler transforms all of this C++ code into a machine code. **Header files do not get compiled just cpp files**. Every cpp file will compiled into something called an **object** file. After that we need some way to stich them together into one exe file and that's is where the linker comes in. It takes all of the obj files and links them together.

How the Compiler Works?

What is a C++ compiler actually responsibel for?

So we write our C++ code as text and that's all it is. It's just a text file and then we need some way to transform that text into an actual application that our computer can run. In going from that text form to an actual executable binary we basically have two main operations that need to happen. One of them is called **compiling** and one of them is called **linking**.

The only thing that C++ compiler actually needs to do is to take our text files and convert them into an intermediate format called an object file. Those object files can then be passed into the linker. The compiler actually does several things when it produces these object files. Firtly it need to pre-process our code which means that any preprocessor statements get evaluated then and and there.

Once our code is proccessed we move onto more or less tokenizing and parsing and basically sorting out this "english" C++ language into a format that compiler can actually understand and reason with. This basically results something called abstract syntax tree (AST) being created which is basically a representation of our code but as an abstract syntax tree. The compiler job at the end of the day is to convert all of our code into either constant data or instructions.

Once the compiler is created this abstract syntax tree it can begin actually generating code. This code is going to be actual machine code that our CPU will execute.

Every single .cpp file that our project contain we actually tell the compiler - Hey compile this CPP file - every single one of those files will result in an object file. These CPP files are things called **translation units** essentialy.

In C++ there is no such thing as a file. A file is just a way to feed the compiler with source code.

A translation unit will result in an object file.

Let's talk about a bit of pre-processing. During the pre-processing stage the compiler will basically just go through all of our pre-processing statments and evaluate them. The most common pre-processor statement what we have is the `#include` . This a really simple statement. You basically specify which file you want

to include and then the pre-processor will open that file, read all of its content and just paste it into a file whrere you wrote your include statement. And that's it.

Variables in C++

We want to be able to use data. Most of what programming is actually using and changing data. We want to be able to read and write from datas so we need to store this data in something called a variable. They allow us to name a piece of data that we store in memory. When we create a variable it is gonna be stored in memory in one of three places **stack**, **heap** or the **static/global** store.

A variable is a name which is associated with the value that can be changed. For example when I write `int i = 10;` here a variable name is *i* which is associated with value *20*. `int` is a data type that represents that this variable can hold integer values.

C++ requires that all variable are declared before they are used.

Syntax of declaring a variable in C++

`dataType variableName = value1;`

for example:

`int num = 10;`

In the example above we declaring a variable called `num` and also defining its value to 10; Declaring a variable means you just create a variable with a specific name, but don't assing its value. Defining a variable means you assign a value to an existing variable.

```
#include <iostream>

int main()
{
  int i; // declaration
  i = 1; // definition
}
```

Let's take a bit closer look on that snippet. How many steps it needed? Firstly we created a the variable and after then assigned its value to it. That is two steps. From C++11 we have something called **list initialization**.

`int i{1};`

This will take only one step to create a variable and initialize its value. And it is more safety like assigning a value to a variable.

```
#include <iostream>

int main()
{
  int i = 1.1; // implicit type conversion from double to int, the .1 will be chop
ped off
  int j{2.2}; // compile time error -> type 'double' cannot be narrowed to 'int' i
n initializer list
}
```

Types of variables

Variables can be categorized based on their data type. For example, in the above example we have seen integer types variables. Following are the types of variables available in C++.

- `int` - these type of variables holds integer value.
- `float` - single precision floating point value.
- `double` - double-precision floating point value.
- `char` - holds character value like 'c', 'F' etc.
- `bool` - holds boolean value *true* or *false*.
- `string` - array of characters.

Types of variables based on their scope

Before going further lets discuss what is scope first. When we discussed the *Hello World Program* before, we have seen the curly braces in the program like this:

```
int main()
{
  // some code
}
```

Any variable declared inside these curly braces have scope limited within these curly braces, if you declare a variable in `main()` function and try to use that variable outside `main()` function then you will get a compilation error.

`used of undeclared identifier 'yourVariableName'`

Now that we have understood what is scope. Lets move on to the types of variables based on the scope.

- Global variable
- Local variable

Global Variable

A variable declared outside of any function (includeing `main()` as well) is called global variable. Global variables have their scope troughout the program, they can be accessed anywhere in the program, int he `main()`, in the user defined function, anywhere.

Global Variable Example

Here we have a global variable `myGlobalVar`, that is declared outside of `main()`. We have accessed the variable twice in the `main()` function without any issues.

```
#include <iostream>

int myGlobalVar = 0;

int main()
{
  std::cout << "My variable is " << myGlobalVar << std::endl;
  myGlobalVar = 1;
  std::cout << "My variable is " << myGlobalVar << std::endl;
}
```

Output

```
My variable is 0
My variable is 1
```

Local Variable

Local variables are declared inside the braces of any user defined function, `main()` function, loop or any control statements (`if` , `if-else` etc) and have their scope limited those braces.

Local Variable Example

```
#include <iostream>

int myFunc()
{
  int myVar = 1;
  return i;
}

int main()
{
  std::cout << "My variable is " << myVar << std::endl;
  myVar = 2;
  std::cout << "My variable is " << myVar << std::endl;
}
```

Output

Compile time error, because we are trying to access the variable `myVar` outside of its scope. The scope of `myVar` is limited to the body of function `myFunc()`, inside those braces.

So can global and local variables have same name in C++?

Lets see an exmaple for that.

```
#include <iostream>

int myVar = 0; // global variable

int myFunc()
{
  int myVar = 1; // local variable
  return i;
}

int main()
{
  std::cout << "myFunc var " << myFunc() << std::endl;
  std::cout << "Global variable " << myVar << std::endl;
  myVar = 123;
  std::cout << "myFunc var " << myFunc() << std::endl;
  std::cout << "Global variable " << myVar << std::endl;
}
```

Output

```
myFunc var 1
Global variable 0
myFunc var 1
Global variable 123
```

As you can see that when we changed the value of `myVar` in the `main()` function, it only changed th value of global variable `myVar` because local variable `myVar` scope is limited to the function `myFunc()`.

Store in variable

You can store your own specific value in a variable like -

```
#include iostream

int main()
{
    // greet the user
    std::string name;
    std::cout << "Give me your name: ";
    std::cin >> name;
    std::cout << "Welcome, " << name << std::endl;
}
```

Note that the `std::cin` will read your input only for the first white space character. When you want to read a whole line use the following syntax -

`std::getline(std::cin, name);`

Operators in C++

Basic arithmetic operators are:

- `+`
- `-`
- `/`
- `*`

To know more about operators in C++, please visit this link

Expressions

Expression is anything that yields a value. Most statements in C++ are expressions.

Expression describe the action to be performed and consist of an operator and one or more operands.

```
#include <iostream>

int main()
{
    int x;
    int y;
    int res;

    std::cout << "Enter 2 numbers: ";
    std::cin >> x;
    std::cin >> y;

    result = x + y;

    std::cout << "The sum is: ";
    std::cout << result;
}
```

Lets reduce the lines of this code. How?

```
#include <iostream>

int main()
{
    int x,y; // no res needed, you will see why
    std:: cout << "Enter two numbers: ";
    std::cin >> x >> y; // read as many variable in one line as you want
    std::cout << "The sum is: " << x + y; // x + y will be evaluated first and the
n sent to cout
}
```

Conditional Statement

Conditional Expression

`a > b` or `c == 4` -> evaluates to *true* or *false*.

Note: non-zero always converts to *true*, zero converts to *false*.

The `if` statement.

The conditional expression evaluated by the if construct must be enclosed in brackets.

```
if (conditional expression)
    statement1; // if the expression is true
else
    statement2; // if the expression is false
```

You do not need brackets until your statements are single commands. You have many options how you will write your `if` statement. For example -

```
if (conditional expression)
    statement1;
else
    statement2;

if (conditional expression)
{
    statement1;
}
else
{
    statement2;
}

if (conditional expression) statement1;
else statement2;

if (conditional expression) { statement1; }
else { statement2; }
```

It is up to you to make your decision. All the four will do the same.

the `if-else` statement

When you want to check more expression.

```
if (conditional expression)
    statement1;
else if (conditional expression)
    statement2;
else if (conditional expression)
    statement3;
.
.
.
else
    statementN;
```

The ternary operator. It is the same as a normal `if-else`, it just have different syntax.

```
conditional expression ? true brach : false brach;
```

A Looping Statement

The `while` statement iterates while the condition is *true*

```
while (conidtional statement)
    statement;
```

The statement may be a compound or a block. **Remember** you do not need brackets until your statement is a single command.

```
while (conditional expression)
{
    statement1;
    statement2;
    ...
    statementN;
}
```

Unlike the `if` statement, which simply switches program flow, the `while` statement iterates around a series of statements while a condition remains true. as soon as the condition changes, the loop is terminated.

Example for a simple `while` statement -

```
#include <iostream>

int main()
{
    int number;

    std::cout << "Enter a number between 5 and 10: ";
    std::cin >> number;

    while(number < 5 && number > 10)
    {
        std::cout << "Incorrect number. Please enter again: ";
        std::cin >> number;
    }

    std::cout << "Your number is " << number;
}
```