**Functions**

Functions in C++ are very similar to functions in Java and C# with one exception. In C++, we can have functions that are not linked to a class or object, in other words, free floating functions. This difference allows for the programmer to create functions without needing a class to hold the function data, which can be very useful for quick functions that can compress your code. You can make classes that contain these functions if you want, but it is not necessary. This also means we can create a number of functions in a program and call them directly, which is a very common feature with certain libraries, such as OpenGL and the Simple DirectMedia Layer (SDL) framework.

**Return Type**

Each function has a type that it returns. This can be a class, an int, an enum, a float etc. It can also be of type void, which essentially means the function returns nothing. Unlike Java and C#, classes returned are local copies of the class, though this can be changed manually and is covered in a future worksheet.

**Function Name**

The function name is the name the function is given. The compiler will usually complain if the name is already in use in the program, so we cannot create two functions in the same scope called ‘foo’ for example. Make sure to give your functions their own unique name.

**Arguments**

A function in C++ can take any number of arguments of any type. Arguments are separated by a comma. Similar to the return type, passing a class makes a local copy, so passing a class to a function and making changes does not affect the original. We will cover pointers and references later which help us solve this problem.

For now we will look at the following example:



Let’s break this down; the return type is void meaning the function returns no value when it is finished, the name of the function is ‘i\_am\_a\_function’, and it takes three arguments in this order, an int, a float and a char. Inside the function the first argument is called ‘arg1’, the second is called ‘arg2’, and the third is ‘arg3’. If we wanted to perform some maths with the first argument, we would access it by calling ‘arg1’.

Let’s look at another example:



Here we have a function with a return type bool, a name of ‘return\_function’ and no arguments. As this function returns a value, there needs to be a return statement somewhere inside it. The compiler will complain if there is not and won’t compiler the program. Note that the return type can be a different type if they can be implicitly converted. For example, while the above function returns a bool, we could return the return statement with:



This only works as the compiler can convert an int to a boolean. Any int value of 0 corresponds to false, whereas any non zero value corresponds to true. Most C++ base types (bool, int, float, char, double) are implicitly convertible among themselves in this sense.

**Calling a Function**

To call a function is very easy. Will simple use the function name and fill in the arguments to the function with data. The arguments can be hard coded data, or can use variable we have defined before:



**Function Prototypes**

When we write code in C++, the compiler reads the file from top to bottom. If it comes across a variable before it has been declared and initialised, the compiling process will usually fail. With functions we have to declare them before we use them. For example, if we want an addition function to be used in my main, we need to declare it before main like so:



When the compiler reads this, we have told it the function ‘addition’ exists and what is does before we call it. The compiler is happy and all is well. Try moving the function beneath main and see what happens. The compiler will complain that the function has not been defined. In this example, the fix is easy; we move the function above main so it is declared first. But what happens if we have a number of functions that call each other?



‘func\_a’ needs to have ‘func\_b’ to be declared above it, but ‘func\_a’ needs ‘func\_b’ to be declared first. This is where Function Prototypes come into play. We can say a function exists, before we state what the code is.



The top line is our prototype. We say there is a function in the program called ‘addition’ that returns an int and takes two ints as arguments but we do not define the code inside the function. We do this later, after the main.

This can fix the problem we had before, by creating our prototypes, then defining the functions later. As the compiler is told these functions exist beforehand, it doesn’t matter what order we define the code in:



Exercises