Portfolio Report 1: Introduction to C++ programming

Jake Spiteri

2020

C++ Basics

This section will cover the most basic aspects of C++.

C++ files are saved with the extension `.cpp` and then compiled with a compiler such as `g++` in Linux.

Let's create a basic 'Hello World' file. We use a text exitor to save the following C++ code as `hello.cpp`.

```
#include <iostream>

int main()
{
  std::cout << "Hello from C++" << std::endl;

  return 0;
}
```

Let's analyze the above code in order to gain some insight as to how C++ works.

- `#include <iostream>` — This tells the program to include the `iostream` header file, which contains a list of functions, classes, and objects which extend C++. In this case we require `iostream` to use `std::cout` and `std::endl`.
- `int main()` — This line initializes the `main` functions and declares that it will return an integer. Every C++ program contains a main function which will return 0 if there are no errors; it is the first function which will be ran when the program is ran.
- `std::cout` — This command tells the program to print things which are pushed into it. Variables such as strings can be pushed into `std::cout` using the `<<` operator. After printing in C++ we often want to start a new line, which can be done using `std::endl`.

In the above program we print 'Hello from C++' by pushing the string into `std::cout`.

We then need to compile the code. In this case we will use `g++`. If we are working in the same directory as the C++ file, then we can compile the file by running the following code in the terminal.

```
g++ hello.cpp -o hello
```

The above code tells the computer to use `g++` to compile the file `hello.cpp`, and output the resulting program as `hello`.

We can then run the compiled program using the command

```
./hello
```

```
## Hello from C++
```

The above code tells the computer to look in the current directory for the file `hello`, and run it.

We now look at defining variables in C++. We save the following code as `variables.cpp`.

```
#include <iostream>
#include <string>

int main()
{
    std::string a = "Hello!";
    std::string b = "Look at this integer:";
    int c = 25;

    std::cout << a << " " << b << " " << c << std::endl;

    return 0;
}
```

The above code imports `iostream` but also `string`, which allows us to define variables of the type string using `std::string`. Similar to before, we have a `main` function. In this case we define two string variables `a` and `c`, and an integer variable `b` which we then push sequentially into the `std::cout` command. Similar to before we can compile the code and run it using the following.

```
g++ variables.cpp -o variables
./variables
```

```
## Hello! Look at this integer: 25
```

Loops

We often run into situations in which we want to use for loops, similar to how we use them in R. The syntax for C++ is as follows

```
for (initialize; condition; increment)
{
  // code which will be run for each iteration of the for loop
}
```

We can write a simple for loop which outputs the 8 times table as follows

```
#include <iostream>

int main()
{
  for(int i=1; i<=12; i++)
  {
    std::cout << i << " times 8 is " << i*8 << std::endl;
  }

  return 0;
}
```

We save the above code as `loop.cpp`, and compile it, and run it below.

```
g++ loop.cpp -o loop
./loop
```

```
## 1 times 8 is 8
## 2 times 8 is 16
```

```
## 3 times 8 is 24
## 4 times 8 is 32
## 5 times 8 is 40
## 6 times 8 is 48
## 7 times 8 is 56
## 8 times 8 is 64
## 9 times 8 is 72
## 10 times 8 is 80
## 11 times 8 is 88
## 12 times 8 is 96
```

Functions

To define a function using C++ we use the following syntax.

```
return_type function_name (arguments)
{
  // computations

  return return_value;
}
```

where

- `return_type` is the type of value that the function returns.
- `funtion_name` is simply the name of the function which we will use to call the function.
- `arguments` are the arguments which the function requires. Remember that we must declare their type such as `int a`.
- `return return_value` returns `return_value`.

Let's define a very simple function which multiplies two doubles. We will multiple 1.5 and 4.

The file `mult.cpp` containts the following code which defines the function `mult()` which is then called in the program's `main()` function.

```
#include <iostream>

double mult(double a, double b)
{
  double c = a*b;

  return c;
}

int main()
{
  double a = 1.5;
  double b = 4;

  std::cout << a << " " << "multiplied by" << " " << b <<
    " " << "is" << " " << mult(a,b) << std::endl;

  return 0;
}
```

We compile and run the program.

```
g++ mult.cpp -o mult
./mult
```

```
## 1.5 multiplied by 4 is 6
```

Multiple file programs

We cannot call a function without first defining it. In some cases we may need to call a function before it is defined, and in this case we can simply declare the function first. It is often best practice to keep a separate header file (with extension `.h`) which declares all of our variables, and include this header file at the beginning of each `.cpp` file.

We will rewrite the `mult.cpp` file such that it includes a header `mult.h` which declares the function `mult`. We will call this new file `mult2.cpp`.

The contents of our header file is

```
#ifndef _MULT_H
#define _MULT_H

double mult(double a, double b);

#endif
```

The beginning of the header contains 'header guards', which check whether the header has been included yet — this prevents us from accidentally including the header multiple times.

The `mult2.cpp` then contains

```
#include <iostream>

// include the header file with function declarations
#include "mult.h"

// define main function
int main()
{
  double a = 1.5;
  double b = 4;

  std::cout << a << " " << "multiplied by" << " " << b <<
    " " << "is" << " " << mult(a,b) << std::endl;

  return 0;
}

// define the mult function after main
// this is okay as mult has already been declared

double mult(double a, double b)
{
  double c = a*b;
```

```
  return c;
}
```

When we compile the code we simply specify both files, as follows:

```
g++ mult2.cpp mult.h -o mult2
./mult2
```

```
## 1.5 multiplied by 4 is 6
```

Variable types and scoping

When defining a variable we must specify what class of data the variable can hold. For example `int a` means that the variable a can only take on integer values. If we were to set `int a = 1`, we could change the value of `a` by setting `a = 2` for example. However, if we try to change the value of `a` to a float such as `a=1.5`, we get an error.

We can perform *type conversion* in C++. This means that if we define a variable `float a = 1.5`, we can convert the class of data held in `a` so it can be held by a different variable. For example, we could then set `double a1 = a`. We could also set `int a2 = a` but this is an unsafe conversion as data is lost — `a2` will only retain the integer part of `a`.

A variable in C++ must be assigned a type which it will retain for the entirety of its lifetime, which can be defined by its scope. The scope of a variable is the area in which it is defined and accessible. The scope is controlled by curly brackets `{}`. A variable is only accessible within the brackets it is defined in, and within and nested curly brackets.

This means that a variable can be defined multiple times throughout the program, as long as the definitions are in different scopes.

A short example is below which is saved to a `scoping.cpp` file.

```
#include <iostream>

int print_a()
{
  int a = 1;

  return a;
}

int main()
{
  int a = 2;

  std::cout << "Within the scope of the main() function, a = " << a << std::endl;

  std::cout << "Within the scope of the print_a() function, a = " <<
    print_a() << std::endl;

  return 0;
}
```

```
g++ scoping.cpp -o scoping
./scoping
```

```
## Within the scope of the main() function, a = 2
## Within the scope of the print_a() function, a = 1
```