2G03 Homework 2, 2020

Due end of day, Thursday, September 24

This document is intended as a tutorial to accompany your lectures in 2G03. As such, we won't get into the nuts and bolts of how C++ does what it does. Instead this document dives into the basics of the Hello World example that everyone should be familiar with from class (if you're not sure what I'm talking about review the "Compiling Hello World" lecture from September 15th). We're going to write an analogous program step by step and go over each of its components. If you get stuck on your assignments, reviewing these tutorial documents is a good place to start. The first time you review this document you're encouraged to code along. We expect you to code along with each question. **You should write a copy of all the code in this document along with the exercises for full credit**. Learning to code is like learning to play an instrument, you have to practice!

2G03's official coding language is C++. Note that much of what is here is also legal C code. The features not present in C are namespaces, iostream, std::cout and std:cin.

1 Writing to the Terminal

1.1 The `main()` function

The `main()` function is the heart of every C++ program. Every C++ program must have a `main()` function. Let's repeat that one more time. *Every* C++ program *must* have a `main()` function. When we compile our program, the `main()` function is the set of instructions that the computer will run. You might have written a great number of wonderful functions, but if they aren't called in the `main()` function (or called by a function that's called by the `main()` function), nothing will happen!

Login to phys-ugrad and make a new directory to work in (if you're unsure how to do this, go back and review homework 0). **Now use your favourite text editor (e.g. gedit) and open a new text file, marcantony.cpp by typing**,

```
gedit marcantony.cpp &
```

Now define the main function as follows,

```
int main()
{

  return 0;
}
```

So what does all this mean? We'll see later that function definitions in C++ always follow the format,

$\langle \textit{function type} \rangle \quad \langle \textit{function name} \rangle () \{ \textit{code} \}$

As you may have guessed, the `main()` function is always called `main` (no exceptions). When we say *function type*, we mean the type of variable that the function spits out at the end. In general this could be anything (or it could be nothing). The type of the main function is always `int` (`int` for integer - the reasons aren't important here).

Compile this program by typing,

```
c++ marcantony.cpp -o MarcAntony
```

You can run the program with `./MarcAntony` **(nothing should happen). Compile the same program but remove the semi-colon after return. The error message you see will come up a lot. Copy and past it into a file called errors.txt for future reference.**

1.2 Including Input and Output

Now that we have our `main()` function ready to go we need to print to the terminal. Doing this from scratch is a herculean task for a new programmer, but luckily for us some angel wrote a bunch of functions for reading and writing to the terminal and put them in a library called `iostream` (which stands for *input/output stream*).[1] In order to use this library we need to include it, which we can do by adding. In C++, which is a superset of C, this is accomplished by the `iostream` library,

```
#include <iostream>

int main()
{

  return 0;
}
```

1.3 Writing to the terminal

In our C++, writing to and reading from the terminal are accomplished using *streams*. A stream is just an object in C++ that stores a location for data to be written. A stream could be, for example, a file. The standard stream for writing to the terminal is `cout`, and feed the stream using the `<<` operator. For example (**Add these changes to your code**),

```
#include <iostream>

int main()
{
  std::cout <<"Friends, Romans, countrymen, lend me your ears\n";
  return 0;
}
```

[1]Technically the function definitions aren't in `iostream`, they're actually in a *source* file which has already been compiled into machine code on the phys-ugrad server (or on your laptop when you installed the C++ compiler). The `iostream` file contains the information about those functions that C needs to use them (i.e. the function declarations or function prototypes), all of this is irrelevant for the current example, but we'll learn about these details shortly and it pays to mention them here. If this seems confusing just remember this, a *declaration* tells C++ *what* a variable or functions is (its type and the arguments it needs), a *definition* tells C++ *how* that function works. Declarations say what, definitions say how (**repeat that three times**).

You might be wondering why we prepend `std::` to the front of the `cout` stream. This is our first example of a *namespace*. We won't encouter namespaces very much in this course, but it's helpful to know about them. Sometimes we write function and variable names that are so sensible and generic that another programmer may well have used the same names. For example, perhaps we wrote a code that takes the absolute value of a number and called it `abs`. How is C++ supposed to know the difference between our `abs` and the C++ `abs` function? When this happens it's dramatically called a name collision. Luckily the C++ `abs` function (and the `cout` stream for that matter) is hidden away in a *namespace* called `std`. Think of it like a family name which helps us tell the difference between all the James's in the world. Of course it would get tedious prepending `std` to everything, so in those cases we use the line `using namespace std;` which tells C++ to assume that any function name we're using might be coming from the `std` namespace and to not worry about it. Using this trick we can **rewrite the above code**,

```
#include <iostream>

using namespace std;

int main()
{
  cout << "Friends, Romans, countrymen, lend me your ears\n";
  return 0;
}
```

It is instructive to attempt to compile the code *without* the namespace declaration, i.e. try compiling,

```
#include <iostream>

int main()
{
  cout << "Friends, Romans, countrymen, lend me your ears\n";
  return 0;
}
```

Your compiler will be quite cross with you, and should tell you that it's never heard of `cout` before. **Read the error message carefully and try to interpret it. This will help you debug your code in the future. Add the error message to errors.txt.**

2 Variables

Now that we understand the `main()` function we briefly review how variables are defined. In the technical jargon C and C++ are both *statically* typed languages. This means that when we define a variable, perhaps a number that we're going to compute, we have to tell the computer both the name of the variable (preferably something descriptive), and the type of the variable (whether it's a word, an integer, a real number etc...). One might ask, are integers not just subsets of real numbers? Why can't C and C++ just treat all numbers the same? The answer is that treating all numbers the same would be inefficient. An advantage of statically typed languages is that compiler doesn't need to figure out how much memory to reserve for each variable. Below is an example of three different variable definitions in C++. **Make a file called variables.cpp and write the below code into that file.**

```
int main()
{
  int a = 5; // an integer
  double pi = 3.14; // a real number

  return 0;
}
```

Compile the code using the command,

```
c++ variables.cpp -o Var
```

Now remove the `int` specifier for the variable `a` and try compiling. Record the error message in errors.txt

For storing actual words and sentences, the C++ language offers the `string` library, which can be included analogously to the `iostream` library by adding

```
\noindent #include <string>
```

At the top of the main function file,

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  // a string of letters
  string greeting = "Friends, Romans, countrymen, lend me your ears!!\n";
  cout << greeting;
  return 0;
}
```

Modify `marcantony.cpp` so that it uses a string variable. Compile and run the code

3 Reading from the terminal

Now that we understand how to construct variables and write to the terminal, we can think about how to read things that the user might need to tell us. Let's consider a simple example where we read two numbers and return their sum. **In a file called `sum.cpp`, write the following code**

```
#include <iostream>

using namespace std;

int main()
{
  int a,b;

  cout << "Please give me two integers.\n";
  cin >> a;
  cin >> b;

  cout << "Your sum is:" << '\t' << a+b << endl;

  return 0;
}
```

Compile and run the code above. What happens if the user inputs the numbers 1.3 and 3.6?

Let's put all this together and write some basic programs. **Complete the following two excercises**

1. Write a program that takes in three integers and returns the product and sum.
2. Read about the `string` type in the C++ documentation. Write a program that takes two words from the user and returns a single string with the second word following the first. For example your program could take "Goodbye" as its first word and "Moon" as its second word.