C++ Basics

Here is the general format for a C++ program:

File: program.cpp

```
/* Comments */
// More comments
#include <iostream>
#include <string>
…                 // Provides access to standard code libraries

#include "myclass.h"    // Provides access to user-defined code

int g;                  // Global variables defined here

using namespace std;    // Defines the namespace we're using

// Define any functions here that are available globally, e.g.
void someFunc() {
      // code
}

int main(char *argv[], int argc)          // Parameters optional
{
      // Code here for main returning an int value
}
```

Later on we will give the format to define classes. For now let's see some basic innards that make up a typical C++ program.

Primitive Data Types: Declaring variables with primitive data types is dust like Java and C. There are some differences with scoping that we will discuss later when we discuss classes and objects

Typical data types: char, int, long, float, double, bool (not boolean)

C++ treats 0 as false and non-zero as true

They behave the same as you would expect, just like in C or Java. This means there are the same pitfalls. For example, if we divide integers:

double x = (2 / 3) * 6;

gives us 0 because we do integer division for 2/3 which results in 0 * 6.

We can also have roundoff errors to watch for. For example:

double d1 = 0.3;

```
double d2 = 0.1+0.1+0.1;
if (d1 == d2)
        cout << "Same" << endl;
else
        cout << "Different" << endl;
```

The new C++ 11 standard has a new data type called "auto". If we use this as the data type then C++ will try to deduce what the type is based on the expression on the right hand side of the assignment.

For example we could write:

```
double x = 3.15 * 3;
auto y = x * 10;
```

This makes y into a double because the compiler see's that x is a double and sets y accordingly. For the primitive data types this is not particularly useful, but when we get to really long data types using templates then the auto notation because much cleaner.

To compile a program using C++11 we have to add the **`-std=c++11`** flag when using g++. This is for version 4.7 or higher. Older versions of g++ require **`-std=c++0x`**. Even older versions don't support C++11 at all. If you are using OS X you can try using clang++ as your compiler.

Here is an example from the command line of tikishla:

- g++ test.cpp –std=c++11

C++11 also has a way to retrieve the data type of a declared variable. It is decltype:

```
decltype(x) y = x * 10;
```

This retrieves the type of x (a double) and creates y as a double.

Later we will see interesting uses for auto and decltype with functions, and especially when we get to talk about templates and classes. Until then, you are probably better off using the old format of specifying a primitive data type.

Strings:
In both C++ and Java we can hard-code a string using double-quotes. However, C++ and Java natively treat strings differently. Java has a built-in string class. Core C++ strings are defined as arrays of char. The null character \0 is used to terminate a string. C++ does include a Standard Template Library (STL) implementation of a "string" data type that behaves similarly to a Java string. (The STL library includes common data structures, such as a vector, hash table, etc.)

To use the STL strings, we must #include <string> at the top of each file that wishes to use strings. Here is an example:

```
#include <string>
using namespace std;
int main()
{
  string s;       // Note lowercase string, not String
  s = "hello";
  s = "hello" + "there";      // Concatenation
  s.length();                 // Returns # of chars in the string
  s[0];                       // References character 'h'
  s[1];                       // References character 'e'
  return 0;
}
```

Output: To print data to the console, use:

```
cout <<  data_to_print;
```

For example:

```
cout << "hello world";              // Outputs hello world
cout << s << " " << x+3;            // Outputs string s concatenated with x+3
```

To print a newline at the end of the output use endl:

```
cout << "hello world" << endl;
```

We can also include the various escape characters (\n, \r, \", etc.) in the string.

Input: To read data from the keyboard, use cin. For example:

```
int x;
double d;
cin >> x;                   // Reads an integer from the keyboard into x
cout << x << endl;          // Output what the user entered
cin >> x >> d;              // Input into multiple variables
```

If we are inputting into a string, this doesn't quite behave as expected if there are spaces in the input. cin only inputs data to the first whitespace, so if we have input with spaces we don't get the desired result:

```
string s;
cin >> s;                   // User types "hello world"
cout << s << endl;          // Outputs "hello"
cin >> s;                   // Reads in the "world" part
cout << s << endl;          // Outputs "world"
```

To avoid this problem we can use various functions such as getline() to input a string to the newline. We'll discuss this later.

Boolean expressions, arithmetic operators, relational operators: Just like Java and C

If statement: Just like Java and C.
One exception from Java is that any non-zero value is considered to be true, while zero is considered to be false. So we could make a statement such as:

```
if (1) { … }   which amounts to :  if (true) { … }
```

Assignment statement: Just like Java and C
One common pitfall is confusing = with ==. While Java will flag this as an error, C++ will not because it is considered legal. An example is below:

```
int i=0;
if (i=1) { …  }
```

The body of the if statement will always be executed because in the expression "i=1" we assign the value 1 into i, and the value tested by the if statement amounts to if (1) {…} As we have seen this is considered true, so we will execute the boy of the statement and at the same time variable i is set to 1.

For loop, while loop, do-while loop, break, continue: Just like Java and C

Defining functions: Mostly like defining a method in Java and a function in C

We'll cover differences in passing by value or reference later, but the general way we define and use a function is pretty close to what you are used to.

Essentially we have:

```
<return type> functionName(parmType parmName, parmType, parmName…)
{
        … code …
        return (something of type <return type>;
}
```

It is typical to define the function prototype, or function header, somewhere near the top of the program. This tells the compiler that the function exists so we can reference it before the implementation may be defined. The function prototype is basically the first line of the function, defining the return type and parameters. You don't need to put

variable names in for the parameters, but most people put them there for a little extra information.

For example:

```
int addOne(int y);                  // function prototype

int addOne(int y)
{
        y++;
        return y;
}

int main()
{
        int x = 3;
        cout << addOne(x) << " " << x << endl;          // Outputs 4  3
        return 0;
}
```

C++ lets us use the keyword const to make things constant and unchangeable, but it can be used in many different ways that can be confusing.

```
const int x = 3;        and
int const x = 3;
```

both make x a constant that is an integer set to 3. You can't change x. In general, the thing to the left is what the const applies to, unless there is nothing there, in which case the const applies to the thing to the right.

We will see how const is used with functions/methods later and that is where it starts to get confusing.

Basically we have covered a good chunk of the "C" part of C++. Here is a sample program that illustrates most of the points covered so far. Given a distance in miles and a time in minutes and seconds, it calculates the speed/pace to cover that distance in minutes/mile. Once again everything should look pretty familiar!

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

using namespace std;

int getMinutesPace(double distance, int minutes, int seconds);
int getSecondsPace(double distance, int minutes, int seconds);

int getMinutesPace(double distance, int minutes, int seconds)
{
      double totalMinutes = minutes + (seconds / 60.0);
      double minutesPerMile = totalMinutes / distance;
      return (int) minutesPerMile; // Truncate anything after decimal
point
}

int getSecondsPace(double distance, int minutes, int seconds)
{
      double totalMinutes = minutes + (seconds / 60.0);
      double minutesPerMile = totalMinutes  / distance;
      // Get what is after the decimal point and multiply by 60
      return (minutesPerMile - (int) minutesPerMile) * 60;
}

int main()
{
  string firstName;
  char c;
  double distance;
  int minutes, seconds;

  do
  {
      cout << "What is your first name? " << endl;
      cin >> firstName;
      cout << "How many miles did you travel? ";
      cin >> distance;
      cout << "How many minutes and seconds did it take? ";
      cin >> minutes >> seconds;

      cout << firstName << ", your pace is " <<
            getMinutesPace(distance, minutes, seconds) <<
            " minutes and "
            << getSecondsPace(distance, minutes, seconds)
            << " seconds per mile." << endl;

      cout << "Do again? " << endl;
      cin >> c;
  } while (c=='y');

  return 0;
}
```