C++ Introduction

RoboJackets Training – C++ - Weeks 1 & 2

Basic Interaction

Before we get into all of the fun and interesting parts of C++ programming, it's important to know how to talk to your program. This will help us double check what the computer's thinking as it runs our code and help us save useful data from our C++ applications.

Standard interaction between C++ and users happens through *streams* that connect to files or the console. *Streams* are “serial” containers that can wrap any data storage mechanism. Being “serial” just means that you can only read the next incoming character. You don't get to skip ahead to read the middle or the end. Objects can push data into a stream, and other objects can read data from that stream, but only in the order in which it was inserted.

Console Streams

The two most frequently used I/O streams in a basic C++ application are **cin** and **cout**. Defined in the **<iostream>** header in the **std** namespace, these streams offer access to the standard console for a quick, text-based user interface. A program can print data to the console using the stream insertion operator (<<) on the cout object.

```
#include <iostream>
int main() {
     std::cout << “Hello, C Plus Plus!” << std::endl;
     return 0;
}
```

Data can be read from the console through the cin object. C++ will automatically parse the text to give you the best data for the type you request.

```
#include <iostream>
using namespace std;
int main() {
     int age;
     cout << “How old are you?” << endl;
     cin >> age;
     cout << “You are “ << age << “ years old!” << endl;
     return 0;
}
```

File Streams

Moving data to and fro the console is great, but sometimes we want to save a lot of data from our program to run some other post-processing on it or further analyze it in some way. C++ makes this easy by giving us streams specifically for reading and writing to files! Defined in the **<fstream>** header, we can use **ifstream**, **ofstream**, and **fstream** to interact with files. i**fstream**s give read-only access to files, **ofstreams** give write-only access to files, and **fstream**s can do both reading and writing.

The following example reads a list of numbers from 'numbers.txt', adds 10 to each, and writes the list to 'output.txt'.

```
#include <fstream>
using namespace std;
int main() {
     ifstream inFile(“numbers.txt”);
     ofstream outFile(“output.txt”);
     while(inFile.is_open() && !inFile.eof()) {
          int number;
          inFile >> number;
          number += 10;
          outFile << number << endl;
     }
     inFile.close();
     outFile.close();
     return 0;
}
```

A common and helpful format for writing data to a text file is the Comma Seperated Value (*.csv) format. This format is compatible with nearly all spreadsheet applications, making it super convenient for cases where you want to easily graph data from your code. Here's an example of how to write data to a CSV file.

```
#include <fstream>
using namespace std;
int main() {
     ofstream outFile(“ouptut.csv”);
     if(!outFile.is_open()) {
          cout << “Could not open output.csv” << endl;
     }
     for(int x = 0; x < 10; x++) {
          outFile << x << ',' << (x*x) << endl;
     }
     outFile.close();
```

```
        return 0;
    }
```

Working With Data

Primitive Types

C++, like most programming languages, has a standard set of “primitive datatypes” built-in to the language for handling basic kinds of data. All other types are derivatives of these fundamental types. Table 1 shows the primitive types of C++ and their corresponding bit sizes.

Type	**Size (bits)**
bool	1
char	8
int	32 or 64
float	32
double	64
void	*

Table 1. C++ primitive types and their sizes. Void is a type that means “no data” and is most frequently used to show when functions and methods take no inputs or return no output.

Sign-age

C++ distinguishes between variables that can hold positive and negative numbers and those that can only hold positive numbers. A “signed” variable can store numbers with both a positive or negative sign. An “unsigned” number has no concept of positive/negative; thus, it can only hold positive numbers.

All primitive types default to their signed forms, but can be manually declared as “unsigned” using the **unsigned** keyword before the type name.

```
unsigned int a = 10;
```

For more information on how negative numbers are handled behind the scenes, and why code like the following example compiles, see the Wikipedia entry for Two's Complement.

```
std::string str = “Test code.”;
unsigned int index = str.find('c');
// How am I comparing an unsigned integer to a negative number?
if(index == -1) {
        std::cout << “'c' not found in “ << str << std::endl;
    }
```

Objects in Memory

An interesting experiment that demonstrates some fundamental concepts of how your computer is organizing your code's data in memory involves the byte size of an object. What would `sizeof(Square)` return for the definition of Square given below?

```
    class Square {
        double width, height;
        int r, g, b, a;
        // Only method declarations following
        …
    }
```

The correct answer is 32 bytes. This is because C++ only needs to keep track of the contents of an object's member variables, and will stack those contents together in memory. Table 2 shows how C++ would store a Square's data in memory.

Bytes	Variable
1-8	width
9-16	height
17-20	r
21-24	g
25-28	b
29-32	a

Table 2. The memory footprint of an example Square object for C++.

Here's another experiment: what does `sizeof(std::string)` return? The answer is 4! How could any string of characters ever created by a man, woman, child, animal, or machine fit into 4 measly bytes??? The answer lies in the almighty pointer.

Pointers

Your computer's memory is a very long list of 1-byte cells, each of which has a numeric address. These addresses helps the computer keep track of where it saved all of your variables as your code runs. For normal variables, your computer just gives a particular cell a label, and knows to go to that cell every time that variable's label is used. A **pointer** is just a variable that contains the address of another variable. In this way, we say that the pointer variable “points to” the other variable.

Here are some important operators related to C++ pointers:

- Address of (&) – gives the address of the given variable in memory.

  ```
  int* B = &A;
  ```

- Derefernce (*) – returns the variable saved at the given address.

  ```
  int C = *B;
  ```

- Arithmetic (+, -, ++, --, etc.) - moves the address this pointer points to in steps equal in size to the size of the type of the pointer.

  ```
  int* C = &A;    // C is 1000
  c++;            // C is 1004
  ```

Pointer arithmetic, combined with what we know about C++ memory footprints, gives rise to a very powerful way to work with arrays. Arrays in C++ are really just a bunch of variables of the same type stored one immediately after the other in a block of memory. To use an array, we just need the address of the first element and some way to know where the end of the array is. This is usually either a pointer to the address after the last element, the number of elements in the array, or a special element value that marks the end of the array (eg. A “null-terminated string”). To iterate through the array, we can just increment the pointer and dereference it to get the element at the next spot. In fact, the standard array access operator is really just shorthand notation for pointer operators!

```
// The following lines are the same
arr[i] = 10;
*(arr+i) = 10;
```

This should now explain the strange char** notation often seen in the arguments for the main() method of many C++ programs. This is a pointer to a pointer.

```
int main(int argc, char** argv) { }
```

Let's break it down. The last asterisk makes argv a pointer to the preceding type. In this case, that type is char*, which is just a pointer to a character. Remembering that pointers can represent arrays, char* is an array of characters, aka a string. Likewise, char** is an array of arrays of characters, or an array of strings. But wait, to use a pointer as an array, we have to somehow know where it ends! That's what argc is for! argc is the “arguments count,” and argv is the array of “argument values.” argc holds the number of character arrays there are in argv, and the end of each of these character arrays is marked with a “null character,” which is just the value 0.

Reasons for Pointers

We now understand some of the mechanics of pointers and what they actually mean, but we don't really know why they exist. Here is a classic example, the swap method:

```
void swap(int a, int b) {
     int tmp = a;
     a = b;
     b = tmp;
}
```

As it's defined above, the swap method does absolutely nothing. This is because C++ arguments are copied when you call a function. So the a variable that swap is manipulating is at a different place in memory than whatever value you passed in to swap as the first argument. The a and b that swap is working with are correctly swapped, but your variables haven't changed. What if we could tell swap where to find our copies of these variables, so it could manipulate those instead? Oh wait, we can! We can ask our variables for their addresses in memory and give those addresses to the swap method as pointers. That looks like this:

```
void swap(int *a, int *b) {
     int tmp = *a;
     *a = *b;
     *b = tmp;
}
```

In this version of the swap method, we take the value at address a and copy it to a temporary variable. Then, we copy the value at address b into the space at address a. Finally, we copy the value in the temporary variable into the space at address b. Now, when we return, we've successfully swapped the variables our caller gave us, and not some local copy of those variables. Of course, we have to do something special when calling this version of swap to make sure we give it addresses instead of regular variables:

```
int a = 10;
int b = 31;
swap(&a, &b);
```

This example demonstrates the main idea behind pretty much every use of pointers, namely sharing specific locations in memory with different parts of our code. We'll return to pointers and their use cases in a bit.

C++ References

Pointers are incredibly powerful and, as we showed in our swap example, can make many useful algorithms possible in C++. With this power, however, comes a great deal of danger. We are essentially being given the ability to edit the raw memory of our computer. This memory holds not just our variables, but also our actual code and that of the operating system! Flip the wrong bit and we could accidentally put our computer in a world of hurt. To grant some of the powers of pointers without as much risk, C++ introduced reference type variables.

Reference types in C++ allow parts of our code to share an address in memory without actually telling each other what the address is. Changing the value of a reference variable will change the value of the original variable, but there is no way to change where your reference variable is pointing to or perform any arithmetic on the address to move through memory. This actually simplifies the syntax of our swap method quite a bit.

```
void swap(int &a, int &b) {
     int tmp = a;
     a = b;
     b = tmp;
}
int main() {
     int c = 10;
     int d = 31;
     swap(c, d);
}
```

Because references are safer than pointers, it is a good rule of thumb to chose a reference type over a pointer type variable whenever you can.

Classes

C++ was originally named “C with Classes,” so it's about time we actually got to making some classes! Classes are the “blueprint” for objects, telling the computer what data is needed to represent an object's state and what code is needed to define an object's behavior. Declaring a class is pretty straight forward.

```
class TestClass {
public:
     int a;
     int b;
     int getNumber() {
          return num;
     }
private:
     std::string secret;
     int num;
};
```

There are a few important things to note here. First, watch out for that semi-colon after the declaration. You'll probably forget it a lot if your IDE doesn't add it for you. Secondly, checkout those weird access modifiers! C++ uses clustered access modifiers. This means that everything that comes after “public:” is public until another modifier is declared. So, in our example, a, b, and getNumber() are all public while secret and num are private.

Headers and Implementations

Classes can be declared in one file just fine, but C++ developers usually break them up into two files to help organize our code a little better. These files are called the header file (*.h/*.hpp) and the implementation file (*.cpp). There's nothing really special about these file extensions, they are purely to help human coders know what to expect in each file. A *.h file hold the class declaration without any definitions for methods and the *.cpp file will hold all of the definitions of those methods.

```
// TestClass.h
class TestClass {
     public:
     int a;
     int b;
     int getNumber();
private:
     std::string secret;
     int num;
};
```

```
// TestClass.cpp
#include “TestClass.h”
int TestClass::getNumber() {
     return num;
}
```

*.hpp files will often be used for utility classes that are small enough to fit comfortably and readably in a single file. The .hpp extension lets the developer know not to go looking for a companion .cpp file, as no such file exists.

The reason for this separation is two-fold. First, it helps developers reading through existing code by giving them a place to quickly glance at the public interface for a class without having to scroll through hundreds of lines of method implementation code. The second part of the reason is related to the actual function of the #include statement.

#include actually copies the entire contents of the given file and pastes them into the current file in place of the #include line. After all of these copies are finished, individual source files are compiled and linked. If classes were all defined in a single file and included every time they were needed, our code would take forever to compile one massive main object file and would have to recompile all of your code every time you changed anything. By breaking classes into headers and implementations, your compiler only has to recompile code that uses your class when its interface changes. Implementations can change as much as they want without triggering a recompile for any code that uses that class.