A Crash Course in C++

Most people taking CS193D will already have some experience with C++. This handout is a quick refresher in case it's been a while. If you need more detail, read Chapter 1 of the book (also coincidentally titled *A Crash Course in C++*).

An Annotated Hello World
The following dead simple C++ program shows all of the components of a C++ program:

```
// helloworld.cpp

#include <iostream>   // C++ standard headers don't use ".h"

/* main() in C++ takes argc (the number of arguments) and
   argv (an array of arguments) */
int main(int argc, char** argv)
{
  // "std" is the standard C++ namespace, like a Java package
  std::cout << "Hello, World!" << std::endl;

  return 0;
}
```

Namespaces
Namespaces, like packages in Java, address the problem of naming conflicts between different pieces of code. For example, you might be writing some code that has a function called `foo()`. One day, you decide to start using a third-party library, which also has a `foo()` function. The compiler has no way of knowing which version of `foo()` you are referring to within your code.

Namespaces solve this problem by allowing you to define the context in which names are defined. To place code in a namespace, simply enclose it within a namespace block:

```
// namespaces.h

namespace mycode {
    void foo();
}
```

```
// namespaces.cpp

#include <iostream>

namespace mycode {

    void foo() {
        std::cout << "foo() in the mycode namespace" << std::endl;
    }
}
```

To call the namespace-enabled version of foo():

```
mycode::foo();    // calls the "foo" function in the "mycode" namespace
```

To avoid being explicit about the namespace with every call, use `using`:

```
#include "namespaces.h"

using namespace mycode;

int main(int argc, char** argv)
{
    foo();  // implies mycode::foo();
}
```

Namespaces are a great example of the difference between a hacked together C++ program and a production quality program. The programs you write should be namespace-savvy.

Preprocessor Directives

Java programmers may not be familiar with the notion of a preprocessor. A preprocessor is like a compiler that runs over the code before the real compiler does its work. In C and C++, lines that begin with # contain commands for the preprocessor. In C, the preprocessor is often used to create faux constants and gross inlined functions known as *macros*. These preprocessor features are present in C++ but are rarely used.

The main use of the preprocessor in C++ is the #include mechanism. Unlike Java, C and C++ functions are *declared* separately from their *definitions*. The declaration (which for C++ classes is confusingly referred to as a *class definition*) is usually placed in a file that ends in .h. Typically, C++ header files also make use of the preprocessor to make sure

that they are only included a single time. Each header defines a unique symbol and will skip its content if the symbol is already defined:

```
// myheader.h

#ifndef __MYHEADER_H__
#define __MYHEADER_H__

// content of the header goes here

#endif // __MYHEADER_H__
```

Worth noting: In C++, the standard language headers are included using angle brackets (< and >) and do not traditionally use ".h", as shown above with <iostream>. Why not? Well, all of the standard C headers are also available within C++. When using the C headers, you *do* include the .h.

Variables, Types, Conditionals, Loops, etc.
Most of the basic types and other constructs are similar between C, Java, and C++. For example, the following snippet of code works in all three languages:

```
  int result = inValue;

  for (int i = inValue - 1; i > 1; i--) {
    result *= i;
  }

  return result;
```

Dynamically Created Arrays and Heap Memory
Recall that standard arrays live on the *stack* and their size is determined at compile time:

```
int* myArray[30];
```

To create an array whose size is determined at runtime, you declare it on the *heap* by allocating new memory:

```
int* myArray = new int[arraySize];
```

Heap memory must be released manually by calling delete. When releasing memory that was allocated an array, you must use the bracket version of delete:

```
delete[] myArray;
```

Heap memory is also used in C to achieve pass-by-reference. We will get into pointer mechanics later in the quarter. For now, we don't need to use pointers for pass-by-reference because we have…

References

A reference is basically a short-hand for a pointer without the messiness of dereferencing or the ambiguity of ownership. Java doesn't have an analogous concept because object arguments to Java methods are passed by reference automatically. Since Java has an object class for every corresponding basic type (e.g. Integer for int), there is always a simple way to achieve pass-by-reference.

In C++, the & character is used to indicate that a variable is a reference. For now, we'll just focus on references as parameters to functions and methods. There are some subtleties we'll get into later.

The first version of addOne() below does *not* use a reference, so the variable that is passed in remains unchanged. The second version uses a reference, so the underlying variable is actually modified within the function. References can be used for basic types as well as more complex types, like classes.

```
void addOne(int i)
{
    i++;  // has no real effect since this is a copy of the original
}
```

```
void addOne(int& i)
{
    i++;  // actually changes the original variable
}
```

Strings

C++ programmers make use of both traditional C-style strings (arrays of characters with a null terminator) and the C++ string class. The string class works much like the Java String class, which is to say that it behaves like you would expect:

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

using namespace std;
```

```
int main(int argc, char** argv)
{
    string str1 = "Hello";
    string str2 = "World";
    string str3 = str1 + " " + str2;

    cout << "str1 is " << str1 << endl;
    cout << "str2 is " << str2 << endl;
    cout << "str3 is " << str3 << endl;

    if (str3 == "Hello World") {
        cout << "str3 is what it should be" << endl;
    } else {
        cout << "hmmm... str3 isn't what it should be" << endl;
    }

    return 0;
}
```

Classes

For people who don't have much experience with Object-Oriented Programming, we'll be talking more about classes in (no pun intended) an upcoming class. Until then, just think of classes as similar to C structs with associated functions. Here is the syntax for a class definition:

```
// Calculator.h

class Calculator
{
    public:                    // external code can call these methods
        Calculator();          // constructor
        ~Calculator();         // destructor

        int   add(int num1, int num2);           // method
        float divide(float numerator, float denominator);    // method

        bool  getAllowNegatives();               // method
        void  setAllowNegatives(bool inValue);   // method

    protected:           // external code can't access these members
        bool  fAllowNegatives;                   // data member
```

```
};    // note the semicolon at the end!
```

The functionality, or *methods,* of the classes are defined as shown:

```
// Calculator.cpp

#include <iostream>
#include "Calculator.h"

Calculator::Calculator()
{
    fAllowNegatives = false;  // initialize the data member
}

Calculator::~Calculator()
{
    // nothing much to do in terms of cleanup
}

int Calculator::add(int num1, int num2)
{
    if (!getAllowNegatives() && (num1 < 0 || num2 < 0)) {
        std::cout << "Illegal negative number passed to add()" <<
std::endl;
        return 0;
    }

    return num1 + num2;
}

float Calculator::divide(float numerator, float denominator)
{
    if (!getAllowNegatives() && (numerator < 0 || denominator < 0)) {
        std::cout << "Illegal negative number passed to divide()" <<
std::endl;
        return 0;
    }

    return (numerator / denominator);
}

bool Calculator::getAllowNegatives()
```

```
{
    return fAllowNegatives;
}

void Calculator::setAllowNegatives(bool inValue)
{
    fAllowNegatives = inValue;
}
```

Finally, here's how other parts of the code use and interact with the class:

```
// CalculatorTest.cpp

#include <iostream>
#include "Calculator.h"

using namespace std;

int main(int argc, char** argv)
{
    Calculator myCalc;  // stack-based Calculator

    myCalc.setAllowNegatives(true);
    int result = myCalc.add(2, 2);
    cout << "According to the calculator, 2 + 2 = " << result << endl;

    Calculator* myCalc2; // heap-based Calculator

    myCalc2 = new Calculator();   // allocate a new object
    myCalc2->setAllowNegatives(false);
    float result2 = myCalc2->divide(2.5, 0.5);
    cout << "According to the calculator, 2.5 / 0.5 = " << result2 <<
endl;

    return 0;
}
```

If You Need More Review…

For most of you, this handout was probably just a refresher. However, if you're coming from a C background or you need a little more help in any of these areas, check out Chapter 1 of the book. It has the same sections with similar examples, but much more detail.