Introduction to C++

Libraries in C++

You will still use #include statements to include libraries in C++, but the libraries themselves will look a little different. In the most recent version of C++, libraries do not have a "`.h`" extension. For example, the C++ input and output functions are in the `iostream` library. To gain access to them, we'd do:

```
#include <iostream>
```

with no `.h` extension.

Namespaces

A *namespace* in C++ is sort of like a package in Java. Both features allow you to separate related classes. In Java, all code that's in the same package is physically stored in a different directory. C++ namespaces do not require this physical separation.

We will not be creating our own namespaces in this class. However, if you're interested, you put:

```
namespace name {

}
```

around the code in each file that you want to belong to the namespace called `name`.

To use code from a different namespace, put:

```
using namespace name;
```

at the top of the file where you want to use the namespace, right underneath the include statements. For example, the C++ I/O functions are in the namespace called `std`, so we would put:

```
using namespace std;
```

at the top of any file with input or output statements.

Using C libraries

You can still include the C libraries in a C++ program, but you need to write the include statements differently. You should drop the "`.h`" extension, and add a "`c`" in front of the library name. For example, we used to do:

```
#include <stdio.h>          //Old way (C)
```

When we wanted to use the C input and output functions. If we wanted to use those functions in a C++ program, we'd do:

```
#include <cstdio>          //New way (C++)
```

IO in C++

While you can still use the C input/output functions like `printf`, `scanf`, and `fgets` in a C++ program, it is considered bad style. C++ has its own input and output functions, so if you want to use C++, you should use these functions instead.

The C++ IO functions are in the library `iostream` and the namespace `std`. If you want to use this functions, add this to the top of the file:

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

Output

C++ has a class called `ostream` which handles output to a buffer. This class defines the overloaded operator <<, which inserts information into the output buffer. There is a special `ostream` object called `cout` that is already connected to standard out. So to print to the console, you will use the `cout` object and the overloaded << operator.

For example, we can print "Hello":

```
cout << "Hello";
```

We will talk more about operator overloading in a few weeks. For now, don't worry how this works – just know that `cout <<` will print text to the console.

If we want to print multiple things, we just use the << multiple times. For example:

```
int val = 4;
char name[10] = "Bob";

cout<<"The value is "<<val<<" and the name is "<<name;
```

Notice that we use another "<<" where we would have used string concatenation (+) in Java. This prints "The value is 4 and the name is Bob" to the screen.

To print a newline in C++, use the `endl` constant:

```
cout << "First line " << endl;
```

```
cout << "Second ";
cout << "Third" << endl;
```

This will print:

First line
Second Third

to the console.

Input

C++ has a class called `istream` which handles intput from buffer. This class defines the overloaded operator >>, which gets information from the input buffer. There is a special `istream` object called `cin` that is already connected to standard in. So to get input from the console, you will use the `cin` object and the overloaded >> operator.

For example, we can ask the user for an integer and then read in the value:

```
int val;

cout << "Enter an integer: ";
cin >> val;
```

This code will read whatever was typed by the user and store it in the `val` variable. Note that we do not pass the address of a variable (`&val`). The operator >> accepts variables by reference, so it can already change their values. (We will talk more about call-by-reference in C++ later on.)

We can also read in several values at once:

```
char name[20];
int age;

cout >> "Enter your name and age: ";
cin >> name >> age;
```

This code will:

1) Skip any leading whitespace
2) Read everything into `name` until it reaches more whitespace
3) Skip whitespace until it reaches a number
4) Read the number into `age`

This code has a similar problem as `scanf` in C – if the user types a name that is more than 19 characters, we will write past the end of the array. We will see a better way to read in strings later in this document.

Classes

C++ is, for the most part, C with classes. In C++ we can now define classes with instance variables, constructors (next time), and functions. This allows us to write programs very similar to what we've done in Java. However, C++ classes look quite a bit different from Java classes even though they have the same functionality.

Here's the format of a C++ class (classes can also have constructors and destructors, which we will talk about next time):

```
class name {
     visibility:
          instance variables;
          function prototypes;
     visibility:
          instance variables;
          function prototypes;
     ...
};
```

The visibility modifiers in C++ are `private`, `public`, and `protected` – just like in Java. A `private` member can only be seen inside the class, a `public` member can be seen wherever the class can be seen, and a `protected` member can be seen inside the class and inside any child classes.

Here is a sample class declaration:

```
class Person {
     private:
          char name[20];
          int age;
     public:
          void setVals(char[], int);
          void print(void);
};
```

Notice that instance variables are declared by listing the variable's type and name. Function prototypes look just like function prototypes from C, except that they are now considered members of this class.

Implementing functions

We can now declare classes, but we haven't actually written any of the code for the class. In C++, class functions are implemented outside the class declaration. Here's the format for implementing a member function:

```
returnType className::functionName(args) {
     //code
```

```
        //return statement, if needed
}
```

For example, here is how we would implement the `setVals` and `print` functions from the `Person` class. Notice that even though we are not inside the class declaration, these functions can access the class's instance variables.

```
void Person::setVals(char n[], int a) {
        strcpy(name, n);
        age = a;
}

void Person::print(void) {
        cout << name << " " << age << endl;
}
```

Creating, using objects
For now, we create objects in C++ in exactly the same way as we declare variables (this will change when we start writing constructors). The way we've declared variables before is with:

```
type name;
```

Like:

```
int val;
```

We will do exactly the same thing to create an object. For example, to create a `Person` object called `p`, we'd do:

```
Person p;
```

In Java, we used a `.` to access methods and variables for a given object. We will do the same thing in C++. For example, we can set our person's name to "Bill" and age to 25:

```
p.setVals("Bill", 25);
```

Similarly, we can print the information about our person:

```
p.print();
```

Example
We've now seen enough to start writing C++ programs with classes, but we still must organize our programs correctly or they won't compile. In this section, we'll look at how to organize, write, compile, and run a full C++ program.

Here are some general rules for organizing C++ programs:

1) Place each class declaration in a separate header file. If the class is called "`Person`", put it in `person.h`. Make sure to include the `#ifndef PERSON_H/#define PERSON_H/#endif` like we did for our multi-file C programs.

2) Place the implementation of your class functions in a corresponding `.cpp` file (NOT `.c`). For example, the functions from the `Person` class should be stored in the file `person.cpp`.

3) Include the header file in the corresponding `.cpp` file. For example, `person.cpp` should have `#include "person.h"`.

4) Create a separate `.cpp` file with a single `main` function. This file must include the header files for every class in your project.

5) Makefiles for C++ programs are similar to Makefiles for C programs. The only difference is that we will now be using the g++ compiler instead of the gcc compiler.

Full C++ program

Let's write a C++ program that uses MVC (model-view-controller) architecture. We will let the user enter 5 numbers, and then we will print the minimum and the maximum of the input numbers. We will use two classes, `Numbers` and `IO`, and a separate `main` function (the "controller" part). Here's the breakdown of the two classes:

Numbers:

```
int nums[5];
void setNums(int[]);
int max(void);
int min(void);
```

IO:

```
int getNum(void);
void printMax(int);
void printMin(int);
```

Here's the implementation of the `IO` class. Notice that we have two files: `io.h` and `io.cpp`:

```
//io.h
#ifndef IO_H
#define IO_H

class IO {
     public:
          int getNum(void);
```

```
            void printMax(int);
            void printMin(int);
};

#endif
```

****

//io.cpp

```
#include "io.h"
#include <iostream>

using namespace std;

int IO::getNum(void) {
     int val;
     cout << "Enter an integer: ";
     cin >> val;
     return val;
}

void IO::printMax(int max) {
     cout << "The max is: " << max << endl;
}

void IO::printMin(int min) {
     cout << "The min is: " << min << endl;
}
```

****

Here's the implementation of the `Numbers` class. Notice that it has files `numbers.h` and `numbers.cpp`:

//numbers.h

```
#ifndef NUMBERS_H
#define NUMBERS_H

class Numbers {
     private:
          int nums[5];
     public:
          void setNums(int[]);
          int min(void);
          int max(void);
};
#endif
```

****

```
//numbers.cpp
#include "numbers.h"

void Numbers::setNums(int arr[]) {
     int i;
     for (i = 0; i < 5; i++) {
          nums[i] = arr[i];
     }
}

int Numbers::min(void) {
     int low = nums[0];
     int i;
     for (i = 1; i < 5; i++) {
          if (nums[i] < low) {
               low = nums[i];
          }
     }

     return low;
}

int Numbers::max(void) {
     int high = nums[0];
     int i;
     for (i = 1; i < 5; i++) {
          if (nums[i] > high) {
               high = nums[i];
          }
     }

     return high;
}
```

****

Now, let's write a separate file with our `main` function. The `main` function will control the flow of the program. Notice that this file must include `io.h` and `numbers.h` since it is using the `IO` class and the `Numbers` class.

```
//main.cpp
#include "io.h"
#include "numbers.h"
```

```
int main() {
      //create objects
      Numbers n;
      IO io;

      int nums[5];
      int i;

      //Get input
      for (i = 0; i < 5; i++) {
            nums[i] = io.getNum();
      }

      //Give input to Numbers
      n.setNums(nums);

      //Give min and max to IO to print
      io.printMax(n.max());
      io.printMin(n.min());

      return 0;
}
```

Makefile

Now, we need a Makefile for our program. The Makefile will look very similar to what we would write for a C program. There are two differences:

1) We will use the g++ compiler instead of the gcc compiler
2) Our `OBJECTS` list will include a `.o` file for every `.cpp` file (in our case, `io.o`, `numbers.o`, and `main.o`).

Here is the Makefile:

```
CC = g++
OBJECTS = io.o numbers.o main.o

nums: $(OBJECTS)
      $(CC) $(OBJECTS) -o nums

clean:
      rm *.o nums
```

When we run:

```
make
```

This will generate the executable file called "nums". So to run our program, we'd type:

```
./nums
```

Strings in C++

Since C++ has classes, there is now a more elegant way to represent strings. C++ contains the class "string", which is very similar to the Java class "String". The string class is backed by a character array, and all memory management of that array is done for us. Here are some examples:

```
string s1;            //s1 = ""
s1 = "hi";            //s1 = "hi"
string s2 = "hello"; //s2 = "hello"
s1 += s2;             //s1 = "hihello"
```

To use the string class, we need to add:

```
#include <string>
```

to the top of our file.

Here are some more examples of what we can do with strings:

```
string str = "hello";

//c = 'e', the character at index 1
char c = str[1];

//c = 'h', the character at index 0
c = str.at(0);

//str2 = "el" (substring of length 2, starting at index 1)
string str2 = str.substr(1, 2);

//str2 = "hello" (copies characters like strcpy)
str2 = str1;

//str = "hhelloello" (insert str2 at position 1)
str.insert(1, str2);

//str = "hloello" (remove 3 characters starting at position 1)
str.remove(1, 3);

//compare str and str2 character by character
if (str == str2) {...}
```

```
//if str comes alphabetically before str2
//can also do >, <=, >=, !=
if (str < str2) {...}

//index = 0, the index of the first "h" in str
int index = str.find("h");
```

Converting between strings and C-strings

Unfortunately, the string class has not completely taken hold in C++ development. Many programs still use character arrays (C-strings) instead of strings. The two types are not equivalent, and so it is necessary to be able to convert between the two. Here's how:

```
string name = "Bob";
char nameC[20] = "Bill";

//toCString now references the C-string "Bob"
char *toCString = name.c_str();

//toCppString now holds the characters "Bill"
string toCppString(nameC);
```

Input

We can get input using a string just like we can a character array:

```
string s;
cout >> "Enter a word: ";
cin >> s;
```

Moreover, this is safer than getting a character array as input. The string class handles all memory management, so it will never write past the end of the array.

Unfortunately, reading in a string or a character array in this manner will only read up until it finds whitespace. This means that you can't read an entire line into a string like you can in Java.

If you would prefer to get your input one line at a time, use the `getline` function:

```
getline(istream, string);
```

This will read an entire line of text from the specified input stream and then store the result in the string variable. For example:

```
string s;
cout >> "Enter some text: ";
```

```
getline(cin, s);
```

Whatever the user types after the prompt (including spaces) will get stored in the variable `s`. (Note: there is no "string tokenizer" for C++. If you want to break apart a line that you read in, you will have to convert it to a C-string and then use the `strtok` function.)

Exceptions

It's important to have some mechanism for handling unexpected states in our programs. C++ has exceptions that are somewhat similar to Java's exceptions. A key difference is that in Java, you can only throw or catch something that has Exception as a base class (like `IOException`, `NullPointerException`, ...). **In C++, ANYTHING can be an exception** – an int, a char, etc. C++ also defines an `exception` base class, and has several common exceptions that extend this class.

This section will not go into much detail about exceptions, but it will discuss how to write a try/catch block and how to throw exceptions.

Try/catch block

The try/catch block in C++ looks similar to the try/catch block in Java:

```
try {
      code that might cause problems
}
catch (type name) {
      handle error
}
```

C++ defines the base class `exception`, so we can write our try/catch block to catch all C++ exceptions:

```
try {
      code that might cause problems
}
catch (exception e) {
      //prints a descriptive message describing the error
      cout << e.what() << endl;
}
```

However, ANYTHING can be thrown as an exception, so this try/catch block won't catch things like ints and chars that are thrown. Here is a try catch block that will catch any type of exception:

```
try {
      code that might cause problems
}
```

```
catch (...) {
      Handle error
      Notice that we don't know what type of error we have!
}
```

Throwing exceptions

Once we learn about inheritance (which won't be until the end of the semester), the best way to throw exceptions would be to define our own more specific exception class that extends `exception` and overrides the `what()` function. Until then, we're better off throwing a string that describes the error, and catching type `char*`. Here's an example:

```
try {
      int num, denom;
      cout << Enter numerator and denominator: ";
      cin >> num >> denom;
      if (denom == 0 ) throw "Division by zero";
      cout << "Division is: " << (num/denom) << endl;
}
catch (char* s) {
      cout << "Exception: " << s << endl;
}
```

If the user enters 0 as the denominator, we will go into the catch block and print "Exception: Division by zero."

If you need to throw an exception in one of your functions, throw a descriptive string.