From Java to C++

Although the two languages are very similar, there are a few differences in syntax and a few more conceptual differences that can catch the unsuspecting programmer by surprise. This document gives an overview of the differences between the two languages and their supporting environments, specifically geared towards students who have completed CSC14400 (Computer Science I) and are now enrolled in CSC24400 (Computer Science II) at Lindenwood University.

1. The Run-Time Environment

In Java, code is first compiled into byte-code and then run through a Java virtual machine; this virtual machine converts byte code into the machine code that the computer system upon which it is running comprehends. A C++ compiler, on the other hand, compiles code directly into machine code. This means that the resulting (byte code) output from a Java compiler can run on any system with a Java virtual machine. However, a compiled C++ program can only run on the type of system it was compiled upon; in order to get a C++ program running on a different system, it may be necessary to re-compile your source code on that different system.

This issue is likely transparent to most programmers, as most programmers are using an Interactive Development Environment (IDE) such as Eclipse or Visual Studio. Such environments hide the details of how to compile and run programs from their users. However, should you find yourself trying to run a program from a command line prompt, be aware that Java required an invocation of the virtual machine with a class name appended (for example: `java HelloWorld`); a C++ compilation will generate an executable that can be run by simply typing its name in (for example: `HelloWorld.exe`).

2. The equivalent of Java's import in C++

Most java code begins by importing various library definitions into a program body through the `import` statement. C++ does not support the import statement, instead opting for the use of the `#include` macro. For example, if you wanted to let the program you are using know that you intend to perform input/output in C++, you would need the `iostream` library, whose details can be disclosed to a program by inserting the line:

```
#include <iostream>
```

at the beginning of the program. Modern C++ compilers also require you to specify a default "namespace" in your program to simplify further usage of this library. For the vast majority of C++ programs, you would follow your `#include` directive(s) with the following line:

```
using namespace std;
```

which gives direct access to global names (such as variables) found in standard system libraries. Other libraries you might want to specify in `#include` directives at the top of a source code file include (but are not limited to):

`iomanip`	allows the user to format input/output
`fstream`	allows usage of file input/output
string	allows usage of C++ strings
`cmath`	allows usage of standard 'C' math functions (like sin(), cos(), ...)
`cstdio`	allows usage of 'C' input/output functions like printf()
`cstdlib`	allows usage of miscellaneous 'C' functions such as rand()

3. The main() function

Both Java and C++ programs require the definition of a single main function, indicating where the code should start executing. However, unlike Java, C++ requires that the main function *not* be part of any class definition. Furthermore, a C++ main function must match one of the following two patterns:

```
int main(void)
{
  ...
}
```

or

```
int main(int numArgs, char *argv[]) // * is not a typo!!!
{
   ...
}
```

You'll see more on what the star means during Computer Science II. An example program including a complete main function can be found in the next section.

4. Input/Output and Files in C++

One of the primary things most programmers want to do is print things to the screen; this is accomplished in C++ through the use of streams. Streams come in two forms in C++:

- **output**, which can send data to a location; the most common named output stream is `cout`, which prints data to the screen when that data is appended to its stream. Items are appended to the stream through use of the << binary operator.
- **input**, which sends data from a stream into a variable; the most common named input stream is `cin`, which takes data typed at the keyboard and places it into variables appended to the input stream. Variables can be appended to the stream through the use of the >> binary operator (note the difference in direction between << for output and >> for input).

A quick example of a complete program in C++ that utilizes such streams follows:

```
#include <iostream>   // allows use of streams and cin,cout

using namespace std;  // makes sure standard cin,cout are used

int main(void)
{
  // prints Hello World to the screen followed by a new line.
  // Note that the endl is a special identifier that explicitly
  // tells the stream to drop down one line.
  cout << "Hello World" << endl;

  float length, width, area;

  // prompt user for input; note the missing endl ...
  cout << "Please enter the length and width:";

  // read in data from user
  cin >> length >> width;

  // calculate rectangle's area using classical formula for such
  area = length * width;

  // output resulting area
  // note that nothing special is needed to spread a command
  // over multiple lines
  cout << "The area of a "
       << length << 'x' << width
       << " rectangle is " << area << endl;

  // main promised to return an int to the caller (the O/S here)
  return 0;
}
```

Streams can also be constructed from files in C++. In order to utilize this feature, you need to include the `fstream` library in your code:

```
#include <fstream>
```

You can then construct a new input stream (similar to `cin`) using the `ifstream` type:

```
ifstream myInputFile("someInputFile.txt");
```

You could also construct a new output stream (similar to `cout`) using the `ofstream` type:

```
ofstream fileForOutput("newFile.txt");
```

Note that, with an ofstream, the associated file will be created if it does not exist or will be overwritten if that file already exists. Using the above examples, a sequence of code utilizing these new streams could be:

```
fileForOutput  << "hello world" << endl;

int sum=0;
int nextVal;
myInputFile >> nextVal;
while (myInputFile) // can use ifstream like bool
{
     sum+=nextVal;
     myInputFile>>nextVal;
}
fileForOutput << "Sum is: " << sum;
fileForOutput.close();
```

5. **Most objects are *not* references in C++**

In Java, all variables representing objects (known as *instances*) actually contain a reference to an object. By default this reference initializes to `null` (i.e. empty), meaning that a programmer must first allocate space for the object before actually using it. For example, suppose you had defined a class called `Coordinate` with (public) attributes `x` and `y` and a default constructor. The following is an example of declaring and using an object of type `Coordinate` in Java:

```
Coordinate MyLocation;

MyLocation = new Coordinate(); // required in Java
MyLocation.x=10;  // without the new, this would crash
MyLocation.y=17;  // without the new, this would crash
```

In C++, however, one does not need to invoke the `new` operator to create a new object (although such can be done using pointers and dynamic memory management - a Computer Science II topic). Instead, when one declares an object, it is immediately given a fixed memory location. Thus, the above Java code's equivalent in C++ might be:

```
Coordinate MyLocation(); // note the constructor invocation
MyLocation.x=10; // without the new, this does NOT crash
MyLocation.y=17; // without the new, this does NOT crash
```

You may also notice that, in the above C++ code, you can invoke a constructor as you declare an object. To do so, you simply place parentheses after the variable name you are constructing and fill those parentheses with the values you wish to pass to the constructor.

It is still possible to create objects that act as references in C++ through the usage of pointers and the `new` operator. Both of these are topics for Computer Science II.

6. **Function Issues in C++**

In Java, all functions are methods of a class, and those methods all have access to attributes of that class. While it is also possible to write such (member) functions in C++ (and such is often done), C++ programmers often write "standalone" functions that are not part of any class. For example, in C++ one could write the following function outside of the context of a class:

```
void mySwap(Coordinate p1, Coordinate p2)
{
  int tempX, tempY;
  tempX=p1.X;
  p1.X=p2.X;
  p2.X=tempX;
  tempY=p1.Y;
  p1.Y=p2.Y;
  p2.Y=tempY;
}
```

If you had two objects of type `Coordinate` called `car` and `truck`, you could then call

```
// note the missing instance variable before the function name
mySwap(car, truck);
```

While such seems simple, there is yet another issue that comes up with passing parameters in C++. Since all objects in Java are actually references, passing an object as a parameter to a function allows the contents of the object to be modified. This happens because the reference to the object is copied as the parameter to the function instead of the contents of the actual object being copied. However, in C++ the actual contents of an object are copied to the function parameter(s) in question. This means that any modification(s) that are made would be made to the copy and would not be made to the actual calling parameters. Thus, in the above example, the values of `car` and `truck` will not change! Fortunately, C++ has a mechanism to force parameters to be passed by reference instead. In the function header, a programmer simply prefixes each parameter name with an ampersand (&) to indicate that each such parameter should be passed by reference. So, the example above would be changed to:

```
void mySwap(Coordinate &p1, Coordinate &p2)
{
  int tempX, tempY;
  tempX=p1.X;
  p1.X=p2.X;
  p2.X=tempX;
  tempY=p1.Y;
  p1.Y=p2.Y;
  p2.Y=tempY;
}
```

Note that the *only* location that the ampersand should be added is in front of the associated parameter in the function header; the body of the function does not need any additional syntax when using the same parameter(s). Also note that it is *not* a requirement that all parameters be call by reference - a programmer could have some be call by reference and others be call by value. Also, the actual function call does not change in any way. Although the example above shows a “standalone” C++ function, parameters to class methods (member functions) can also be passed by reference by using the ampersand.

One final note on functions - arrays are automatically passed by reference in C++; there is no need to specify the additional ampersand to ensure that such happens for array based parameters. More details on arrays in C++ can be found below.

7. **Global Variables in C++**

Just as functions can be declared outside of classes in C++ (but not in Java), variables can be declared outside of classes in C++ (but not in Java). Such variables are considered *global* in scope, meaning they exist in the entire program. This is essentially the same scoping rule(s) as existed for Python and JavaScript (but did not exist in Java.) For example, consider the following C++ program:

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

// global variables
int one;
int two;
int three;
int four;

void myFunc(int three)
{
  int one = 11;
  two = 22;
  three = 33;
  four = 44;
  cout << "one=" << one << ", two=" << two
       << ", three=" << three << ", four=" << four << endl;
}
int main(void)
{
  one = 1;
  int two = 2;
  three = 3;
  four = 4;
  cout << "one=" << one << ", two=" << two
       << ", three=" << three << ", four=" << four << endl;
  myFunc(333);
  cout << "one=" << one << ", two=" << two
       << ", three=" << three << ", four=" << four << endl;
  return 0;
}
```

The resulting output would be:

```
one=1, two=2, three=3, four=4
one=11, two=22, three=33, four=44
one=1, two=2, three=3, four=44
```

8. **Arrays in C++**

While arrays are objects in Java, they are not considered objects in C++. Thus, creating an array does not utilize the `new` keyword in C++[1]. For example, consider the following array definition in Java:

```
double myJavaArray[] = new double[25];
```

An equivalent array definition in C++ would be:

```
double myCPPArray[25];
```

Note that, since arrays are not objects in C++, you can *not* access properties of arrays (through dot-property or dot-method notation).

Also, as discussed above, arrays are automatically passed by reference in C++ (without needing to use the & syntax to accomplish such.) Thus, any changes that are made to an array parameter inside of a C++ function are actually being made to the caller's version of the corresponding array.

9. **Elementary Object-Oriented Design in C++**

When defining classes in C++, there are a few minor differences with Java's syntax for the same. These include the manner in which attributes and methods are declared to be public or private and how inheritance is specified.

In Java, every method and attribute must have its declaration preceded by the `public` or `private` (or `protected`) keywords. In C++, such is done in "groups" within a class definition instead. For example, consider the following class definition in Java:

```
public class Demo
{
  public int a;
  private int b;

  public Demo(){a=0; b=0;}

  private int silly(){return a-b;}
}
```

To accomplish the same in C++, one might code the following:

[1] The experienced C++ programmer will realize that you can utilize the `new` keyword when mixing arrays and pointers, a topic that will be discussed in CSC24400.

```
class Demo
{
public:
  int a;
private:
  int b;
  int silly(){return a-b;} // silly is also private
public:
  Demo(){a=0; b=0;}
};
```

Essentially, after one of the `private:` or `public:` lines, all of the attributes or members take on the corresponding access privilege (private or public respectively) until another such line is found (or the end of the class definition is found). Should you fail to specify an access modifier (public or private), the default is private. You may also have noticed that the class itself is not prefixed by public or private; it is assumed to be the equivalent of a public class in Java. You might also have noticed the trailing semicolon (`;`) at the end of the class definition - this is required in C++.

While it is legal to specify a function body in curly braces within the class definition, doing so for longer methods (member functions of longer than 2 or 3 lines) is generally frowned upon in C++. C++ gives the programmer the option of placing just the method's header inside of the class definition and writing the body for that method elsewhere (hidden from a user of the class). This idea will be discussed in Computer Science II.

Inheritance is also supported in C++. You may recall that you could declare a class `CellTower` as follows in Java (using the `Coordinate` class from above):

```
public class CellTower extends Coordinate
{
  // ... add some more attributes and/or methods here
}
```

The equivalent in C++ might be:

```
class CellTower: public Coordinate
{
  // ... add some more attributes and/or methods here
};
```

Note that the `public` keyword on the class header line above indicates that all public members of `Coordinate` should also be public in `CellTower`.