CS225 : Data Structures and Software Engineering
Introduction to C++

Jason Zych

Chapter 1

Introduction to C++

1.1 Why C++?

In this class, the data structures you learn will be taught from a general viewpoint, and the lessons you learn will be applicable regardless of the particular programming language you use. However, we do want to give you actual programming practice with these ideas as well, and in order to do that, we need to choose *some* language to use. So, why C++ and not Java? Well, there are three reasons:

1. First, independent of any particular language merits, knowing C++ will be helpful for you in the future just as knowing Java will be helpful for you in the future. At the completion of the first two software courses – CS125 and CS225 – you will know the two most widely used "new" languages and will be prepared to use either language in future courses and research, and also in industry work (whether job or internship). From that point of view, knowing only Java after the first two software courses would not be as beneficial for you. In addition, C++ is more or less a subset of an older language, C. Certain things you can do in C still can be done in C++, but they tend to go unused in favor of new features. However, the basics are still the same, and since a great deal of today's legacy code is written in C, if you have a project in the future that requires you to know C, you already have a significant head start. So, before we even get into the "learn C++ for what the language itself can teach you" arguments, the case can be made that knowing C++ will likely be quite helpful for you in the future simply due to its usage in much software in the real world.

2. Second, and more importantly, there are features and concepts that are in C++ and not in Java, or that are similar to those in Java but implemented differently, and it is helpful to be familiar with those ideas. In particular, the concept of pointers – an idea not present in Java – carries with it a large number of other concepts that need to be learned, not the least of which is a solid idea of what is *really* going on in the computer's memory as your program compiles and runs. This is an idea that can be glossed over in Java due to Java's high level of abstraction. C++ has a slightly lower level of abstraction in many cases – you *can* program in a high-level of abstraction, but you can also explore more details in an effort to optimize your software. Learning these ideas will give you a better understanding of the interface between programs, compilers, and memory, and therefore will serve as a nice introduction to future courses, where you will be exploring machine architecture and

compilers in greater detail. Further, with every language you learn, the next language becomes easier to learn, and so learning the differences between C++ and Java will not only mean you know C++ for its sake, but it will help build the ability to learn new languages and ideas quickly, which is a skill that is useful in coursework, research, and industry alike.

3. Finally, there is one C++ feature in particular that is very useful for this course in particular. The C++ feature in question is the concept of *templates*, and as the C++ standard has stabilized, the usefulness of templates has continued to increase. In our course, we will be dealing with very generalized, abstract structures – not data types like `Clock` or `StudentRecord`, where you are pretty sure what the data will be, but data types just as `List` or `Tree`, types which leave open questions such as "list of what kind of element?" or "tree of what kind of element?". Once you get used to templates, they are a very helpful tool for this kind of data type, and so in my opinion, C++ fits very nicely with this course, just as Java's relative simplicity makes it a very good fit for an introductory course such as CS125. (In fact, there are rumors that Java may eventually contain a template-like feature, though likely with a simpler syntax than the syntax for C++ templates.)

1.2 Our Java-to-C++ road map

We will be presenting C++ by assuming you have a knowledge of Java – Java being the language taught in the prerequisite to this course here at U of I – and working from there. You don't *have* to know Java to understand our discussions over the next few lectures, since I will be explaining the features of C++ in and of themselves. However, I will also be making comparisons back to Java – not only teaching a C++ feature, but also pointing out how that particular C++ feature is similar to or different from a particular Java feature. This will help you (if you know Java) by letting you think of some of the C++ features as merely slight variations of the Java features you already know, which might make learning the C++ feature easier than it would be if you had to think of the C++ feature as an entirely new concept. So if you don't already know Java you will simply be losing this benefit, that's all.

The conversion from Java to C++ encompasses three different kinds of subjects:

1. There are those language constructs and syntax which are basically the same in C++ as in Java. Most of these constructs are the most basic ideas – language types, control statements (i.e. `if` statements and `for` and `while` loops), the `main()` function, and expression and statement syntax are all more or less the same in Java and C++. The reason for this is because Java was designed based on C and C++ syntax. Where the designers of Java felt C++ syntax could be improved, they tried to improve it, but the most basic ideas in C++ were carried over to Java almost without alteration. For the most part, we will just go ahead and use these syntax constructs without much comment, since you are used to them, but any detailed discussion of them will occur in section.

2. There are those language constructs which implement ideas similar to those in Java but involve either some slight conceptual differences or else syntax which is somewhat different than the syntax in Java that implements a similar feature. Examples of these types of language features include global variables, the I/O facilities in C++, and the usage of

external files and libraries. We will discuss some of these details in lecture and some in section.

3. Finally, there are those C++ concepts which either don't have much of a parallel in Java at all, or else are very different than the similar idea in Java. Examples of these topics include pointers, explicit deallocation of memory, and templates. Into this group we will also place the idea of inheritance. Inheritance is covered thoroughly in CS125 and we are assuming you have seen it before; however, there is enough detail involved in the inheritance syntax in C++ that it's worth discussing it in lecture. In addition, our discussion of C++ inheritance syntax will also serve as a quick review of the actual topic itself – a review that will be helpful due to the difficulty of the topic. The introduction and primary discussion of all of these "big picture" topics will occur in lecture, though we will reinforce some of the ideas in section.

So, in a nutshell, we'll discuss the harder stuff in lecture, and in section you'll cover the easier stuff and also reinforce some of the harder stuff.

One last note before we look at some code – there are some concepts that are the same in C++ and Java but are simply given different names. Among these is the idea of the subroutine. In Java, subroutines were known as "methods", but in C++ they are generally called "functions" instead of "methods". In other languages (such as Pascal) certain functions or methods might be called "procedures". All these terms mean essentially the same thing, and for informal purposes they can be used interchangably.

1.3 Converting a Java class to a C++ class

Before we do anything else, we need to cover the most basic idea in object-oriented programming – the class. The following is a quick "conversion guide" to turn Java classes into C++ classes.

```
Example: a Java class

public class Coord
{
   private double xCoord, yCoord;

   public void Initialize(double xInit, double yInit)
   {
      xCoord = xInit;
      yCoord = yInit;
   }

   public void Reflect()
   {
      double temp = xCoord;
      xCoord = yCoord;
      yCoord = temp;
   }
}
```

What we will now do is convert the above Java class, step-by-step, to a C++ class.

The first step involved is quite simple and yet extremely important: you need to add a semicolon to the end of the class declaration itself – that is, after the closing brace of the class:

```
public class Coord
{
   private double xCoord, yCoord;

   public void Initialize(double xInit, double yInit)
   {
      xCoord = xInit;
      yCoord = yInit;
   }

   public void Reflect()
   {
      double temp = xCoord;
      xCoord = yCoord;
      yCoord = temp;
   }
};
```

This is an almost trivial difference, and thus easy to forget. However, because of the way C++ classes are compiled (you'll get a better sense of why compilation is different in just a second), if you forget the semicolon, the compiler can get very confused and generate all kinds of error messages that say everything *but* "you are missing a semicolon". So, you end up spending all kinds of time tracking down bizarre-sounding errors and trying to figure out what is wrong, but in the end, nothing helps. This is, of course, a situation you would like to avoid. If the compiler simply told you that your problem was a missing semicolon, this difference would be nearly irrelevant, but instead it turns out to be hugely important because of the way in which forgetting it can confuse compilers. This is why we are stressing it as something you absolutely, positively, **cannot** forget to add.

Next, we need to look at access permissions. In Java, each data member or method has its own access permission – you use the keywords "private" or "public" or "protected" before each function and before each line of variable declarations. In C++, we *group* our variable access permissions. That is, we type the word "public" or "private" once, followed by a colon, and then all functions and variable declarations that come between that access permission word and the next one fall into the category of the first access permission word.

So, in our above example, we could simply add the expression `private:` at the top of our class, and then list our variable declarations. This would be followed by the expression `public:`, and then the function definitions. In addition, you don't need to name classes as "public" or "private", so the access permission before the class name itself would not appear in C++.

```
class Coord
{
private:

   double xCoord, yCoord;

public:

   void Initialize(double xInit, double yInit)
   {
      xCoord = xInit;
      yCoord = yInit;
   }

   void Reflect()
   {
      double temp = xCoord;
      xCoord = yCoord;
      yCoord = temp;
   }
};
```

Of course, just as in Java, you can arrange your member data and functions however you like – you don't *have* to have the data first and the functions next.

```
class Coord
{
public:

   void Initialize(double xInit, double yInit)
   {
      xCoord = xInit;
      yCoord = yInit;
   }

   void Reflect()
   {
      double temp = xCoord;
      xCoord = yCoord;
      yCoord = temp;
   }

private:

   double xCoord, yCoord;
};
```

The last change we need to make is also the most involved. In C++, unlike in Java, there is a distinction made between *declarations* and *definitions.* Now, for variables, there isn't much of a distinction to be made. You declare them, and then you use them. End of story. But for functions, the two concepts are more clearly distinct.

Consider, for example, the function `Initialize` which is a member of the class `Coord` above.

```
void Initialize(double xInit, double yInit)
{
   xCoord = xInit;
   yCoord = yInit;
}
```

The first line of the function – the function signature – tells you the information you need to know in order to use this function in your code: the name of the function, the number of parameters it has, the type of those parameters and the order they must be listed in, and finally the function's return type. You need this information if you want to call this function yourself.

However, the code that comes between the braces is different. We don't need to know anything about that code in order to call the function in some other code. Sure, we want to know what the function does, but we could read that from a specification. To actually *use* the function, we need to type a function call into our code, and that means we need to know what name to type, what variables and values we can pass as parameters, and what types of variables we might assign the return value to. But once we have done that, we can compile and run the code just fine. So, assuming we have selected the correct function to use (documentation is an important aid in making that decision), we don't really need to see the code that is used to implement the function. That is, we don't need to see the *definition* of the function, only the *declaration* – the function signature followed by a semicolon.

So, the only thing we want listed in the "`class`" syntax construct are declarations. For this reason, our "`class`" syntax construct is known as a *class declaration.* It is this construct that declares the member data of the class and the member functions of the class. But, the actual definitions of those functions are put elsewhere in the code.

```
class Coord
{
public:

   void Initialize(double xInit, double yInit);
   void Reflect();

private:

   double xCoord, yCoord;
};
```

The above class declaration could then be stored in its own file, called a *header* file. Header files usually have a `.h` suffix, but the actual name of the file can be whatever you want (unlike in Java where it has to correspond exactly to the name of the class). However, it is a good idea to make the file names as helpful as possible, and since `Coord` is a short name, using "`coord`" as the name of the file makes sense. So, the above code could go in a file named `coord.h`.

This raises the question, "what about the implementation code, the function *definitions*?" And the answer is, they are written outside of the class. For example, for the `Initialize` function you would elsewhere have the code:

```
void Initialize(double xInit, double yInit)
{
   xCoord = xInit;
   yCoord = yInit;
}
```

This is not quite right, however, because other classes could have `Initialize` functions, too. So, we use a special syntax operator called the *scope resolution operator* to indicate that *this* particular `Initialize` function is the definition of the `Initialize` declaration for the `Coord` class. The scope resolution operator is typed out as a double colon (::), and it appears after the name of the class and before the name of the function. So, our *real* function definition should appear as follows:

```
void Coord::Initialize(double xInit, double yInit)
{
   xCoord = xInit;
   yCoord = yInit;
}
```

Likewise, the definition for `Reflect` would appear as follows:

```
void Coord::Reflect()
{
   double temp = xCoord;
   xCoord = yCoord;
   yCoord = temp;
}
```

In a sense, the scope resolution operator creates a single name – the name of the function is `Coord::Reflect`, for example, to distinguish it from some other function in some other class, such as `Rectangle::Reflect`.

These definitions tend to all go in one file, usually given a `.C`, `.cc`, or `.cpp` suffix. So, accompanying our `coord.h` file would be a second file, `coord.C`, which would appear as follows:

```
#include "coord.h"

void Coord::Initialize(double xInit, double yInit)
{
   xCoord = xInit;
   yCoord = yInit;
}

void Coord::Reflect()
{
   double temp = xCoord;
   xCoord = yCoord;
   yCoord = temp;
}
```

The "`include`" keyword is something you will learn about in section, but suffice to say here that it is a keyword which allows the `.C` file to have access to the declarations of the `.h` file.

So, to summarize, there are three things you need to do to convert a Java class to a C++ class, and they are, in order of difficulty:

1. add a semicolon after the closing brace of the class declaration,
2. group the access permissions, i.e. remove the "`public`" or "`private`" from in front of each data member and function, and instead list a group of functions or data members after a "`public:`" or "`private:`" declaration (don't forget the colon after the access permission word!),
3. and finally, separate the class declaration and the class definition, placing the declaration (consisting only of data member declarations and member function declarations) inside a `.h` file, and placing the class definition – i.e. the member function definitions – inside a `.C` file and scoping each of them to the class name using the scope resolution operator.

You can then use the class name as a type, just as you did in Java:

```
Coord c1, c2;
```

We will be exploring the syntax of variable declarations and object usage in the next lecture.