Lecture 30

Differences between Java and C++ (cont.)

We finished up our introductory discussion of the differences between Java and C++ using the `first.cpp` file as a reference:

Even in this simple program, there are some noticeable differences:

- no class! C++ doesn't require one (but does have them, of course). Java needs a class no matter what!
- The `main` method has a simpler method header. Also, no arguments, and a return type of `int`. The `int` return type is so an error code can be returned to whoever executed the program. Typically, a 0 is returned if the code ran correctly, and an int value greater than 0 is returned indicating some sort of error. There are other versions of the `main` function in C++ that we'll see later.
- `#include` statements. These may look similar to Java's import statement, but they're really not. They are the equivalent of cutting and pasting the mentioned file at the include location.

  When we say `#include <file>`, we are using a built in system file.

  When we say `#include ''file''` we are referring to a user class file, and we must give an appropriate path name.

  In fact, anything starting with "#" sign in C/C++ is a pre-processor directive. This means that these instructions are performed *prior to* compilation.
- `using std namespace`. This **is** the equivalent of Java's import statement.

  Remember (maybe?) in Java, that import makes our life easier. If we want to use Java's `ArrayList` class, then we can import `java.util.ArrayList`. Now we can declare new variables like this:

  ```
  ArrayList<String> list = new ArrayList<String>();
  ```

  Java imports "java.lang" by default, so we don't have to put it in our code. C++ has no such default. We always have to specify.

  In C++, the `vector` type is in the "std" namespace. By saying

  ```
  use std namespace
  ```

  This is like "import java.util.*;" So we can just create a new vector like this:

  ```
  vector<int> nums;
  ```

  Without the "using" command, we refer to types in the library as follows:

  `std::vector` or `std::cout`

  The `::` operator is known as the *scope resolution operator*.

  The sample file `first_no_using.cpp` gives a C++ file that is equivalent to `first.cpp`, but uses only the scope resolution operator to avoid using the `using namespace` command.
- Function prototypes. In Java, we could place our method definitions anywhere in the class file. C++ requires that we have seen the declaration of a function or variable before we use it, so it must appear before it in the file. There are times when this is difficult, or even impossible (mutual recursion). To handle these cases, C++ allows *function prototypes*. Function prototypes allow us to declare the name before we implement the code, so that we can refer to the function name before we implement it.

- Operator overloading. Notice the "`nums[i]`" notation. `nums` is of type `vector`, which is just a type like the Java `Vector` or `ArrayList`.

  The C++ documentation for `vector` shows the "operator[]" entry in the member function list. This is operator overloading. We have overloaded the [] operator, so that it acts like `at` member function. The `at` function just returns the value at a particular index in the array.

Philosophical Differences

The last item in our list above – operator overloading – is a consequence of the other major difference between Java and C++. We mentioned the compromises and design decisions made in C (and adopted by C++) to make the code run faster. There is also a huge philosophical difference between Java and C++.

In Java, we have a small number of primitive types, and everything else is an object. When we create an object:

```
Object o = new Object();
```

We are creating a name for the object that holds a memory reference, which is a reference to a location in memory where the object exists. Typically, we pass around this memory reference. This is why we can't use "==" with Java objects like strings. We are only testing for aliases, not object equality.

In C++, the designers went to a lot of trouble to make all objects in C++ behave like primitive types. This is very different from the Java approach, and it has consequences.

For example:

- When we declare the vector, we are actually creating a vector, too! This is different for Java, which only creates the reference variable, and requires the new operator to create the object.
- In fact, we are calling the 0 parameter constructor of vector. When we call the 0 parameter constructor, we **DO NOT** use (). This was done on purpose so that default looked like creating a primitive (e.g., int).
- We have similar syntax to Java (the '.' operator), but we also have an array like syntax thanks to the operator overloading. Again, this is to make the `vector` object look and behave like a primitive array type.

Because it wants to treat objects as primitives, the behavior of many similar statements is different if executed in Java versus C++.

Refer to sample file `first_interesting.cpp` from the previous lecture for the following:

In the `moreInteresting()` function, we create a second `vector` called `nums2`, and we set it equal to `nums` with the statement:

```
vector<int> nums2 = nums;
```

What would Java do in this case? In Java, `nums` and `nums2` would both be references to an object. When we say `num2 = nums`, we would actually be copying the reference to the `vector nums` into `nums2`, so that they would both point to the same object in memory.

Thus, in Java, when we say `nums2[0] = 15` (or the java equivalent: `nums2.setElementAt(15, 0);`), we would also be setting the 0^{th} element of nums to 15. Both names refer to the same object in memory.

In C++, this is not the case. We want our objects to behave like primitives, so the = operator makes a copy of the whole object, and stores the copy in nums2!

This gets even more interesting when we look at parameter passing to the function max. Parameter passing in Java and C++ uses pass-by-value. So a copy of the value of the actual parameter is stored in the formal parameter.

In Java, the value is a reference, so the reference is copied, which still points to the same object. If we follow the reference to the object in memory, and change the object, we see the change in the calling method.

In C++ we make a copy of the vector (it is not a reference!!). So a change to the vector in max has no effect on the vector in the calling function.

Classes in C++

Next, we are going to look at defining classes in C++.

Classes are similar to Java, in that we have a class definition, and we label member functions and instance variables as `private` or `public`.

C++ classes are different than Java classes, in that a class is defined by a statement in C++.

Refer to the sample file `intcell_onefile.cpp` from the code page for this lecture.

Here, we have a very simple class definition and usage. The class is called `IntCell`, and it is basically just a wrapper class for a primitive `int` value.

We have to define the class before using it, so we place it above the `main` method. But once we've done that, we can use it as much as we want.

Notice that we have created two `IntCell` objects. Each is a separate object of the class. This is a lot like Java with multiple objects of the same class.

When creating objects of a class in C++, there are some syntactic tricks that are available to make our objects look and behave more like primitives:

- We can use the standard notation for calling the one parameter constructor, or we can use the "=" assignment operator. When creating an object, this is equivalent to passing one parameter in parentheses (if the class lets us!). Note that this can be turned off in the class definition if it is not intuitive to use the one parameter constructor this way.
- We can also omit the "setValue() call, and use the "=" operator again to change the value.

The one file version that defines and uses a class has some definite drawbacks. Most importantly, if we wanted to use our `IntCell` object in another program, we would have to repeat the class definition. This defeats the purpose of creating classes.

The sample files `IntCell_better.h` and `IntCell_better.cpp` provide a slightly better version.

There is no magic here. All we've done is place the class definition inside the `IntCell_better.h` file, and kept the main in `IntCell_better.cpp`. We've also had to add another #include statement. Remember

that this is like cutting and pasting the referenced file into the current file at the location of the #include statement.

Why is this better? Now, another program can include `IntCell_better.h` and use IntCell object as well! This is what we want.

But, it's still not great. If we change the implementation, then we will have to recompile all the files that use it! This may be a lot of work.

It would be even better to separate out the implementation from the declaration of the class.

The sample files `intcell.h`, `intcell.cpp`, and `intcell_tester.cpp` show how we can accomplish this.

We have the declaration of the class in the `intcell.h` file. We have the implementation of the class in the `intcell.cpp` file. And we have a file that uses IntCell objects.

Now, if the implementation changes, only the intcell.cpp files needs to be recompiled. (Although all executables may need to be relinked. This may not be a problem if the objects are dynamically linked to the program, but this is a discussion for another day).

The declaration of the class, without any implementation in file `intcell.h` now looks a lot like a Java interface file. In fact, in functions in a similar (though not the same) way. Programs that want to use our `IntCell` objects will include the header file, which will allow them to refer to all the public member functions listed in the declaration.

There is **one more** thing to worry about. When our projects get large and complicated, it is possible that we are going to have multiple header files including other header files, which include other header files, etc.

It is likely that eventually, we will have the same header file included into the same project more than once. If we just have function prototypes (like we had in our `first.cpp` example) then it is wasteful, and slows down compilation, but it will work.

If we try to include a class declaration more than once, we won't even be able to compile. We will need another pro-processor directive to solve this problem. The `#ifndef` command test to see if a value is defined. If it is not, then all of the code in the `if` block will appear in the header file. If the value is defined (i.e., the test is false), then it will be as if we deleted the code from the source file! This demonstrates how powerful the pre-processor commands can be.

```
#ifndef INTCELL_H
#define INTCELL_H

... body of header file intcell.h

#endif
```

The first time we try to include the file, the value `INTCELL_H` is not defined, so the code in the `if` block is included, and we execute the pre-processor command that defines `INTCELL_H`.

The next time we try to include the header file, the pre-processor test returns false, because `INTCELL_H` is defined, and we don't include the class definition of second time. These are called *include guards*, and are a common technique in C/C++.