Lecture 36

C++ Operator Overloading

We have already seen operator overloading in C++. The operator= (one of the big 3 functions) is just overloading the "=" operator so that the code does the right thing when we try to assign one object to another existing object.

So, what operators can we overload? You can find this information in lots of places, but the Wikipedia page is a nice resource:

```
http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B
```

It turns out, we can overload most of them. Notably, though, we cannot overload "::" and "." These are very important.

Why would we want to overload an operator? Remember that the C++ philosophy is that we want our objects to look and behave like primitives.

To see why this might be desirable, consider a Java example.

```
String s1 = new String("Hello");
String s2 = new String("Hello");
```

If we want to test to see if `s1` is equal to `s2`, we have to use the `equals()` method of one of the strings.

```
if(s1.equals(s2)) {...
```

If we try to use the comparison operator "==" we won't get the answer that we want because we are just comparing the references (which are different).

It would nice for readability if we could say

```
if(s1 == s2) {...
```

and what we got was true if the strings were character for character matches of each other (even if they were not the same object in memory), and false otherwise. This is the idea behind operator overloading. Let's look at a simple example where we overload the "==" operator.

Refer to the `person` files on the code page for this lecture.

We have a very simple `Person` class, where we define a person as a name and a social security number. For simplicity, we implement the class in the header file, instead of separating out the implementation as we normally do.

We may decide that two people are the same if they have the same social security number. Certainly, in a perfect world, this would be true. So, we can write an equals function much like we would in Java. It takes another `Person` object, and compares to the current `Person`. It returns a `bool` depending on the result of the comparison.

We overload the "==" operator by writing a function named "operator==" that basically does the same thing. It takes the same parameter, and has the same return type. In essence, it is the same function just with a special name that is recognized by C++.

In fact, since they are the same, we can just call one from the other. That's it! It works the way we want.

There are some complications, of course, so let's look at a slightly more complicated example.

Refer to the `linkedlistop` files on the code page for this lecture.

Here, we are going to add some operator overloading to our linked list example so that it uses more of an array type of notation.

We begin by adding a function named "operator`[]`" to our class definition. Our first attempt takes an integer value that represents the index of the list value that we would like to reference. Its return value, then, is `int` because it will return the value at that index.

Next, we must implement this function in our .cpp file. We walk down the list until we are at the appropriate node. Then, we return the value stored at that node. Notice, too, that we have added some `assert` statements in case the user asks for an illegal index.

We can run our test program, and the overloaded method works great for getting the value stored at the list at the specified index. However, we would really like to mimic the array notation that we've seen used with the `vector` class.

When we try to use this notation to set a value, we get an interesting error at compile time:

```
operatortest.cpp:17: error: lvalue required as left operand of assignment
```

We have returned an `int` *value* from our overloaded member function. That is, we are using *return-by-value* when we call our function. Logically, it does not make sense to set the value of "5" or something similar. How do we solve this problem?

Remember that previously, we were able to create a reference variable with a declaration like this:

```
int a = 10;
int &b = a;
```

After this reference declaration, anytime we used `b` in the code, it was like using `a`. We can do something similar with the return type for our function.

If we return a *reference* to the value, instead of a copy of the value, then we can use it like we would use the variable itself.

Careful, though, we have to do this correctly. One idea might be to return a reference to the value from the overloaded function in the linked list implementation. **This is wrong**, and won't work. The reason is that

we are using return-by-value from the `value()` function of the `Node` class. So, if we create a reference to this value, it is just a reference to a temporary copy of the value.

What we really need is a reference to the actual `data` member of the node class. We will need to modify the `Node` class to achieve this.

We add a new member function to the node class that returns a reference to the `data` member. Note that the return type is now `int &`. We have to rewrite our operator`[]` function so that it is returning the reference to the value, not the value itself.

Now, when we use our notation on the left hand side of an assignment, it works because it is a reference, not the actual value. It is just like using the name of the private member. It also continues to work on the right hand side because on the right hand side, C++ just evaluates the reference to its value.

C++ Exceptions

Exceptions in C++ can look very similar to Java exceptions, but there are some notable differences.

Refer to file `exception.cpp` on the code page for this lecture.

We'll begin with some standard exceptions. The function `exception1` wraps a `try-catch` block around a memory allocation. We intentionally try to allocate more memory than our system can handle, to see the results.

When we execute the code, we throw a `bad_alloc` exception. The `what` function of the exception gives us the text message that was used to create the exception. We also see that the allocation attempt has caused some other text to be printed warning us of the problems that it encountered.

The most important behavior here, though, is that control flow continues after the exception, as indicated by the fact that the code reaches the print out of "return to main" in the `main` method. This is exactly what we'd expect given our knowledge of Java exceptions.

Next, we examine the function `exception2`. Here we will try to write to a file. We know there are several problems that can occur when performing `I/O`. First, we create an output file stream. Next, we must tell the file what sorts of exceptions to look for while working with the output file stream. This gives us a great deal of control, but is also more work, since the default is to ignore all exceptions.

The different types of exceptions can be or'd together to form a bitmask that will control the behavior. From the documentation, `failbit` indicates "The last input operation failed because of an error related to the internal logic of the operation itself," and `badbit` indicates an "Error due to the failure of an input/output operation on the stream buffer." You can check the documentation for other codes.

We execute the code with no such file on the disk, and the file is created with the output text inside. We execute the program again, and see that another line has been appended to the file. This is due to the `ios_base::app` flag that we passed to the `open` function. Next, we change the file permissions of the file on disk, so that no one can write to the file, and we execute again. This time, we see that we have caught the `ofstream::failure` exception. We can also get the cause of exception with a call to the `rdstate` function. If we check the documentation, we see that the exception corresponded to the `failbit` option.

Again, we verify proper exception handling by noting that the control flow has returned to the `main` function.

Probably the most significant difference between exception handling in Java and C++ is that in Java, you

can only throw objects that extend `Throwable`. In C++, you can throw anything. Exceptions, other objects, primitive values. Literally, any type.

Example `exception3` demonstrates this with a contrived example. First, we call a helper function named `exception3helper`, whose job is to examine an integer value that is passed in, and decide whether or not to throw something.

If the value is too small, we throw the integer value 1. If it is too large, we throw the integer value 2. If the parameter happens to equal 47, we throw a C++ `string` object. Finally, if we don't throw anything, we'll print out a success message.

In the calling function, we place the call to the helper function in a `try-catch` block. Passing different values yields the anticipated results. We must set up different `catch` blocks for each *type* that is thrown. Then we can handle different values of the same type within each `catch` block.

Perhaps most important in this example is the use of the `throw` list in the throwing function. Here, we explicitly list all the types that can be thrown by this function. Note, that if we do not give a throw list, then anything can be thrown without error. This is different from providing an empty throw list (i.e., `throw()`). An empty throw list indicates that the function cannot throw an exception of any kind. If it tries, the `terminate` function is called, and the program ends. Notice that if we force this behavior, the control flow terminates before it returns to `main`.