Lecture 31

Review

Review of what we've learned so far about C++.

- In C++, class definitions are statements.
- They must appear before they are used in a program.
- We can separate out the definition of the class from the implementation of the class.
- We do this by placing the definition with function signatures and instance variables in the header file (.h), and putting the implementation in a separate source (.cpp) file.
- Note that when we implement the class member functions, we need to use the class name and scope resolution operator to refer to which function we are implementing.

C++ objects (as we've seen them so far) behave differently than Java objects.

- C++ wants their objects to look like primitive types.
- The "=" operator actually makes a copy of the object.
- Passing the object as a parameter also makes a copy (pass by value).

`const` operator

The `const` keyword is part of the member function signature. By placing it in the signature of a member function, we are saying that the function cannot and will not change the underlying object. In our `intcell_des.h` example, `getValue()` is guaranteed not to change its `IntCell` object when it is called.

Remember that in Java, we have different types of methods – *accessor* methods and *mutator* methods. Accessor methods access data from the object, and mutator methods change the state of the object.

In C++, we use the `const` keyword to indicate that a member function is an accessor. Member functions that are not labeled are assumed to be mutator member functions.

Since it is part of the signature, we need to include it in the implementation in the `intcell_des.cpp` file.

The `const` requirement is strictly enforced by the compiler. That is, if we try to change the object in a `const` member function, the code will not compile.

C++ seems to care more about whether functions are accessor or mutator methods. This comes from the C++ philosophy that objects behave like primitives. In Java, we can declare that an object be a constant (using the `final` keyword). However, it is only the object's reference that is constant, not the underlying object. In C++, when we declare an object to be constant, we mean the whole object. To enforce this, C++ uses the `const` operator. Thus, if we have a constant object, it will only be allowed to call `const` (i.e., "safe" member functions) to guarantee that it stays constant.

A preview of the destructor method

There is something new in our `intcell_des.h` file that was not in the `intcell.h` file from last lecture. We added a new member function signature that is the name of the class with a tilde in front of it.

We know that the member function with the same name as the class is our *constructor*. The signature with the `~` refers to the *destructor*.

This is something that Java doesn't have, but which C++ requires. The destructor runs when an object is destroyed. It's purpose is to do any cleanup that is required when the object is deleted. For example, deleting (by calling the destructors) of any member objects.

We notice 2 interesting things about our `intcell_des` example:

- We don't call the destructor, it's called automatically for our objects when the `main` function (that is, the function that created the objects) ends.
- The destructors are called in reverse order that the objects were created. Why do you think that is?

Other things to know about the destructor:

- We will almost never call them explicitly. We can, of course, but we will probably not see these cases in this class.
- The destructor does not actually delete the current object, that is someone else's job. It's job is to take care of any thing that it created.

Very soon, we will start talking about the memory management differences between C++ and Java, and we'll start to see why the destructor is necessary.

For now, and for your assignment this week, we will just put an empty destructor in our class. It isn't strictly necessary, C++ will automatically include a default destructor if we don't write one. But we want you to get into good habits, and always explicitly write them.

cassert

We use the `assert` statement in a very similar way to the `assert` in Java. One small difference, we don't have the option of specifying the output that the user sees when the `assert` fails.

Also, we can include the line `#define NDEBUG` in our program, which will tell the compiler to ignore the assert statements.

IMPORTANT: the `#define NDEBUG` line must appear **before** the `#include <cassert>` line. Otherwise, it will not work.

some g++ practice

We didn't get to this in class, but it may be useful.

We have been using g++ in its simplest form so far, so it looks like we just compile by doing the following:

```
> g++ -o name_of_executable first_source.cpp second_source.cpp ...
```

In this way, our source code can be spread across any number of files, and we can combine them into a single executable name `name_of_executable`.

This works, of course, but there is more going on behind the scenes than we may realize.

The first thing g++ does is run the pre-processor. We can do this, too, by running g++ with the "-E" flag.

```
> g++ -E some_source.cpp
```

The dumps the result of pre-processing to `stdout` (i.e., the screen). We can also capture the output in a text file. The result of pre-processing is also a legal C++ source file. We have just performed all of the pre-processor directives in the original file.

The next thing that g++ needs to do is compile the individual source files that it got from the pre-processor. We can do this with the "-c" flag. This will generate an object file, which typically has the extension .o. If we don't specify the name with the -o flag, it will automatically name the file with the same name.

```
> g++ -c first_source.cpp
> g++ -c second_source.cpp
> ls
first_source.cpp first_source.o second_source.cpp second_source.o
>
```

The .o files are compiled code. That is, they are machine code. But, they have not been linked. In class, we have seen several examples where the implementation of a function, and the call to the function were in different source files. After compilation, the code that executes the call is in a different file from the code that holds the implementation of the function. What they share is that they both use the same name for the function.

It is the job of the linker to link the files together. Again g++ will do this for us. We don't need a flag this time, g++ recognizes that it has object files, and knows that the next step is linking.

```
> g++ first_source.o second_source.o
```

The output of the linker is an executable file (assuming there are no errors in our code). We can specify the name of the executable by giving g++ "-o" flag and a file name, as we did above.

If we don't specify an output file name, g++ will create and executable named `a.out`.

We run our executable by specifying the path and executable name. Typically, we will be in the same directory as the executable, but we still specify the path (relatively or absolutely).

```
> ./a.out
```