Lab 10

28 March 2019

This week we'll take a brief break from the `Set` library and revisit a class we saw way back in Lab 4: `Card`, representing playing cards. At the time, we were more focused on learning how pointers work, but today we'll use the class to practice operator overloading and some related ideas.

To start, copy the files from `/home/shared/162/lab10/` into your working directory for this lab. Look at the files; they are similar to the ones from Lab 4 but I've updated a few things. Start writing a readme file and describe what's in the directory according to our usual readme format. Read each file and write down questions about anything you're not sure about.

Vim FOTD: windows

Go into your directory (which you've already copied the lab files into) and type

```
vim -o Card.h Card.cpp
```

Instead of opening all the files sequentially (as would happen if you omitted the `-o`, it opens them all at once! If you use `:q` (which, you'll recall, closes a file without writing it), it just closes one of the windows. (`:wq` also affects only the current window.) To add an extra window, another colon-mode command is handy:

```
:new test_Card.u
```

will open a new window showing `test_Card.u` . You can also try

```
:e Makefile
```

Notice that `:new` opens a file in a new window, while `:e` opens a file in the current window.)

And, of course, when we say "window" here, it is only with respect to the single vim process. All of these "windows" are within a single terminal window. There are two main advantages to using vim windows instead of separate terminal windows to edit multiple files: first, all the switching back and forth can be easily done from the keyboard. And second, they share buffers, so you can hit `10dd` in one window, deleting ten lines, and `p` in another, pasting precisely those ten lines with no funny whitespace or line numbers or tab conversion or anything else.

Vim's window management commands start with Ctrl-W (abbreviated `^W`):

`^W h`	Left one window
`^W j`	Down one window
`^W k`	Up one window
`^W l`	Right one window
`^W w`	Cycle cursor through windows one-by-one
`^W s`	Split current window into two windows (horizontally)
`^W v`	Split current window vertically into two windows
`^W r`	Rotate windows (actually changing their positions)

To quit all windows at once, use `:qa`. (If you've edited some of them, you may need to use `:wqa` or `:qa!` .)

Defining ==

As you saw when you read chapter CI6,[1] and mentioned in class, in C++ you can write method definitions that let you use builtin operators like `==` or `<` with classes that you write.

Go into `Card` and add a method to overload `==` to test whether two `Card` objects are equal to each other. Conveniently (and not coincidentally), there is already an `isEqualTo` method to actually do the work for you. To declare the overloaded operator, you'll declare a method whose name is `operator==` , following the form on p436. When the method is defined, the body can just call `isEqualTo` to do the work.

Don't forget to add a test case to `test_Card` to confirm that this works; note that unlike the name of the method (which will be `operator==`), the name of the test block has to be an identifier, so you'll start it with something like

```
test opEqEq
```

[1]Which I'm not fully recapping here.

So what?

We already had `isEqualTo`, so what benefit is there in overloading `==` ? The convenience of writing something like

```
if (card1 == card2)
```

is nice, of course, but the big win is that a lot of library functions are designed to use an overloaded `==` to do their work. Look inside the file `show_library_stuff.cpp` now. Uncomment the first commented-out part (now that you've implemented `==`). Save the file, run `make`, and then run `./show_library_stuff` to see what it does—it makes use of builtin library functions to do its work, and they make use of the `==` operator to do their work. In many ways, implementing certain operators lets you unlock a whole section of the C++ library.

You'll notice that all the library functions I've used in there take as their first two arguments `cards.begin()` and `cards.end()`. This is one of several uses of the "iterators" that we started to see this week—in this case, a way to tell the library algorithms to process the whole vector, from beginning to end.

Defining <<

The C++ streaming (input/output) operators provide a nifty way for writers of classes to make their objects "read-able" and "write-able"—since `<<` and `>>` are just operators, you can overload them too! When you write something like

```
cout << 5;
```

the thing on the left is an `ostream`, and the thing on the right is an `int`, so C++ looks for and finds a definition for `operator<<` that takes an `ostream&` and an `int`. Deep in the libraries, there is a function defined whose header is

```
ostream& operator<< (ostream& out, int n)
```

to do this work. Since that also returns an `ostream&`, it means that when you write

```
cout << 6 << 7;
```

it performs the first operation (`cout << 6`), which results in an updated `ostream` that can serve as the left side of the next `<<` operator.

In the past, we've made `print` methods that take an `ostream&` to do their work; when we make an `<<` operator, the body of that method will be much like what we've written before; it's really just the headers that will be different.[2]

To make it possible to output your `Card` using streams, you need to do two extra things (that are both specific to the stream operators):

1. In `Card.h`, in the public part of the `Card` class, declare a function called `operator<<` that takes an `ostream&` and a `const Card&` , and returns an `ostream&` , and precede this declaration with the word `friend` so C++ knows that it is a *friend function* and not a *member*—it's not part of the `Card` class but it can look at the private contents of a `Card`—and
2. In `Card.cpp`, write the function. The header will be the same except for the word `friend`. (Notably, it will not include `Card::` anywhere in the line!) Do *not* print anything directly to `cout` here—use the `ostream` parameter instead.

To test this, you can write a test case that creates an `ostringstream` and prints some cards to it, and then verifies that the resulting string is what you expected:

```
ostringstream testout;
testout << queenH << " " << jackC;
check (testout.str()) expect == "QH JC";
```

This is very similar to how the `print` method of `Maze` was tested in the code I gave you for Lab 7.

So what?

The benefits of this one might be a little clearer—once you define a stream operator for a class, you can put objects of that class into the stream just

[2] Again, I won't recap the book here; refer to Section C6.2 for more details.

as if they were built-in types. Once you've got it implemented, uncomment the second part of the demo file. Now that we can conveniently print out `Card` values, you can see the effects of a few other library functions that got unlocked when you implemented `==` .

Side note: once `==` and `<<` are both defined on a type, you can use values of that type directly as `expect` targets in a `.u` file. Remember wanting to write something like

```
check ( Location{3,4}.east() ) expect == Location{4,4};
```

in the maze project? If you were to overload `==` and `<<` for the `Location` class, that line would work exactly as you'd want.

Defining = and the Rule of Three

For a lot of classes, including `Card`, you do not *need* to explicitly overload the `=` operator; if you don't, the compiler will auto-define one that works well in many cases (a fact that we've relied on any time we assigned a `Location`, `Card`, or `Maze` to another variable!).

What does it mean to overload `=`? As the assignment operator, it changes the contents of `this` object to match those of a given other object—similar to a constructor that takes a (const reference to) a given other object, known as a "copy constructor". You really only need to define either one when your class has to manage extra resources (such as `new` pointers, or open files, or similar)—and the Rule of Three says if you think you need to define either one, you should be defining both of them plus a destructor (which takes care of cleaning up the resources when the object goes away).

The `Card` class doesn't need any of that. But file this away for future reference.

Other overloads

Overload `<` (less-than comparison). You should be sure to make aces *high*, that is, an ace is not less than any other card (but all non-aces are less than aces). You can ignore suit for this comparison. Once you've got it done, you can uncomment the last part of the library demo file—you've unlocked the builtin sort function!

Use the same technique you used on `<<` to overload `>>` as well (this time to read from an `istream&`).

Handin

Hand in by 4pm Wednesday, as `lab10`.

Rubric (tentative)

RUBRIC

1	Present in lab with preview stuff done
Defining ==	
1	Header in class definition ♣
1	Body defined correctly
1	Tested in `.u` file (fail ok) ♣
Defining <<	
$^1/_2$	Header correct, `friend` ♣
1	Body defined correctly
$^1/_2$	Tested in `.u` file (fail ok) ♣
Other definitions	
$^1/_2$	< header ♣
1	< well tested including ace logic (fail ok) ♣
1	< defined with correct ace logic
1	>> ♣
$^1/_2$	>> tested (fail ok) ♣

♣ indicates point is only available if the code compiles, with at least a stub for the relevant method(s).

Extras

Go back and revise your `Maze` and `Location` classes to make use of `==` (for `Location`) and the stream operators (for `Maze` and `Location`), and clean up your test files.