Lab 2

4 Sep 2014

This week's lab is a bit thick, but that's because it's designed as a sort of manual for creating classes. Save it! Until you absorb it all (which will probably take at least a few weeks) you'll want to refer back to it later.

Command line feature of the day: permissions

When you first log in, type `ls -al`. (Those are both lowercase L, not numeral 1.) You'll see a listing of the files in your directory, plus listings for the current directory itself (that's the line whose name is just a period—a period by itself refers to the current directory). Each line has extra information about the file, but right now I want you to look at the first part of each line, which might look something like

```
drwxr-xr-x
```

or

```
drwx------
```

The first letter says whether it's a directory (or, occasionally, some more exotic thing), but the other nine report the *permission* settings for the file, and stand for "read", "write", and "execute". The first batch of three indicates the permissions the file's owner has, the second batch is for the file's group (don't worry about this for now), and the third is for the permissions granted to anyone in the whole world (or at least those on the system).

If you type

```
ls -l /home/dblaheta
```

you'll see a listing of my home directory. There is, to be honest, a lot of crap in there, but it shows a variety of permissions. Near the top, `0830-basic.circ` is a plain file with `rw-r--r--` permissions, meaning that I can read and write it but everybody can read it. I have a series of three-digit course directories that have `rwxr-xr-x` permissions, which means that I can

read and write the directories and work with the files in them (although not all of the files inside are world-readable!). My downloads directory is `rwx------`, which means that I can do anything with it but nobody else can.

Now list your own directory again using `ls -al`. What are the permissions on the directory itself? What are the permissions on your files? Depending on just when your account was created and which one of us did it, the default permissions may vary a bit. If you have a home directory with `rwxr-xr-x` permissions, that means that other people can see it; if your home directory is `rwx------`, then nobody can look inside. Some people (like me) prefer the former, generally granting access and only denying permission to more sensitive directories inside; others (like Prof. Marmorstein) prefer the latter. (Try it: when you `ls -l /home/robert`, you won't have permission to do so.)

To grant permission for a directory (such as the current directory), you can

```
chmod o+rx .
```

which grants read/execute permission to "other" people; to lock people out of a directory, you can

```
chmod o-rx .
```

If your home directory is generally readable, you may want to deny other people access to the directories where you keep your coursework.

Note that just because someone else has open permissions on their directory does *not* license you to circumvent a professor's rules on looking at or using other people's work (just as leaving a house's door unlocked does not grant permission for a burglar to go in and steal stuff). The traditional model (open directories, locking individual files) works when a community is based on trust.

Vim feature of the day: `.vimrc`

As you may know, there are a number of commands that you can type into Vim, prefixed with a colon, that cause it to behave differently. For instance, if you type `:set number` it will display line numbers down the side of your screen.

You might also know that if you type those commands into a file named `.vimrc` in your home directory, the commands *always* run, every time you start Vim. This means you can configure Vim to do some pretty powerful things for you. In particular, you can set it up so that Vim does nearly all the work of correctly indenting your C++ code, so that you can read it more easily and so that it better resembles how experienced programmers lay out their code.

Go to your home directory and edit the file (by typing `vim .vimrc`). Into that file, type the following:

```
set cindent shiftwidth=2
set ignorecase smartcase incsearch
set number ruler
set showmatch
syntax on
```

You can find out more about any option in Vim's internal help if you're curious (for instance, type `:help ruler`). If you prefer your indentation a little deeper, you might want to use 4 instead of 2 for your `shiftwidth`. Once you've made these settings (just save the `.vimrc` and quit), when you edit a file you'll be able to use the `=` command to reindent code, e.g. type `==` in command mode to reindent the current line, or `15==` to reindent the next 15 lines. This, together with syntax highlighting, may be among the very most useful of all Vim features.

Designing a data type

In the past you've talked about structs, and may be familiar with something like the following process, but I'm going to formalise a design procedure for classes that addresses first the data, and then the methods. Read the following short description of each step, and then work through the next section to step through a concrete example.

1. Describe the data.

   This can be short, but it's important: what is the coherent description of this data type? Why are its pieces bundled together? This sentence or two will go into a comment above the class declaration, so make it helpful.

2. Give examples.

   What does data of this type look like? Notice that we haven't actually written code yet, so these can just be on paper in whatever form is most convenient to draw them. For a class representing fractions, one example might be $\frac{2}{3}$. There should usually be more than one, although there's no reason to add more for their own sake; each additional example should illustrate some meaningfully different case. Is $\frac{4}{1}$ a valid fraction? What about $\frac{3}{6}$ (as distinct from $\frac{1}{2}$)? Or $-\frac{5}{7}$? These questions can't always be answered in the absence of a purpose for the data; whether unreduced fractions are allowed would depend very much on the application they're used in, for example.

3. Declare the class and instance variables, and

4. Define the (basic) constructor(s).

   These two arguably go together; although one (the instance variables) has to do with internal representation and the other (the constructors) have to do with its external face, at least one of the constructors tends to be pretty straightforward, maybe even obvious, once you know what the variables will be.

5. Encode the examples.

   Once you have constructors defined, you are able to call them and create actual instances of the data type. Don't wait! You should be able to construct every example you wrote out in step 2, and if you can't (or if it doesn't compile), you can go back now and fix the constructors before you get too deep into anything else. For now, you can just construct them in `main`, although we'll see a slightly different way next week.

Trying it with `Location`

We'll now step through the data design process to create a `Location` class. Eventually, we'll want it to be able to represent things like the location of the start and finish in a maze, as well as other locations both inside and outside the maze. (Why outside? Well, you'll need to be able to ask questions like "is this location inside or outside the maze?", which won't work very well if you can't represent locations outside the maze!) Keep that context in mind as you work through this process—it will make a difference in how you think about some of the pieces.

1. Describe the data.

Based on the above paragraph giving the context, write a sentence or two describing what `Location` itself will be responsible for. What pieces of data will it store? What are the data types for those pieces of data? What is the job of a `Location` value?

(Actually do write this out, by hand, in the space above. Don't just read through this part of the lab.)

2. Give examples.

Remember that you're not writing code yet at this point (so no worries about syntax or C++), but give a few examples of `Location` data. Note that it will be useful to be able to represent negative values (although they are outside the edge of the maze).

(Again, *actually write it out* in the space above. Give at least a few legitimate examples of `Location` data. Feel free to make notes about why they are significant and/or interesting.)

3. Declare the class and instance variables

In your directory for this lab, edit a file named `Location.h` that will hold the declarations for this class.

First, set up the header file with the `ifndef` structure you saw in the book. At the very top of the file, type

```
#ifndef _LOCATION_H_
#define _LOCATION_H_
```

and at the end,

```
#endif
```

Every other thing you ever type in this file will go between those.

Next, type in the class skeleton, which looks like this:

```
class Location
{
  public:

  private:

};
```

Precede it with a `/* comment */` containing the description you wrote in step 1.

Finally, in the `private` section, add the instance variable declarations. That is, revisit the examples you wrote in step 2: for each piece of data that makes up a `Location`, declare a variable with the appropriate type and a reasonable name.

4. Define the (basic) constructor(s).

As you saw in the reading, the constructor is how you make sure the instance variables have values when an object is first created. In the `.h` file, we will *declare* the constructor, and in the `.cpp` file, we will *define* the constructor.

Since you're already in `Location.h`, we'll start there. In the `public` part of the class definition, add

```
Location (int i, int j);
```

Note that at this point in the design, the parameter list should include one parameter for every instance variable you declared in step 3. (For future reference: if you need to declare something `explicit` or `const`, or set default parameters, you would do that here as well.)

Save this file and exit the editor. Now edit a file `Location.cpp` to hold the actual method definition. At the top of it, begin with the header:

```
#include "Location.h"
```

Then, type in the constructor definition. Remember that the job of a constructor is to give initial values to *each* of the fields of a struct or class. Since this definition is not inside the class declaration, we need to clarify the name of the method by using the scoping operator `::` and the name of the class. Your code will look a bit like this:

```
Location::Location (int i, int j)
{
  x = i;
  y = j;
}
```

or maybe this:

```
Location::Location (int i, int j)
{
  row = i;
  col = j;
}
```

The important thing to note here is that the assignments in the constructor should be using the parameters to give initial values to the instance variables you declared for the class, whatever you called them. If you're not sure what to write, this would be a good point to consult with the students sitting near you in the lab, if you haven't already. Talk over what you're seeing here, and what you think it does.

5. Encode the examples.

To actually create the examples you wrote by hand in step 2, create a new file `test_Location.u` that has the following contents:

```
#include "Location.h"

test suite Location
{
  fixture:

  tests:

}
```

and after the `fixture` line, declare and init examples of a few `Location` values. For instance, if you wrote "(3,4)" above, you might encode that as

```
Location exampleLoc = Location(3,4);
```

Eventually, when we have methods to test, these examples will be the data we'll use; a "test fixture" is the term generally used to describe the standard batch of data examples used in a particular test suite.

(Sidebar: whenever you're writing a `.u` file, you're using a testing language called Unci that I wrote for intro students using C++ to learn how to write test cases and test suites. Our `compile` command knows about Unci files, and when it is given a `.u` file it does what it needs to do to compile the test—and when you compile an Unci test suite, you don't need to define your own `main` function.)

At this point, you have:

- declared the `Location` class in `Location.h`,
- defined the `Location` constructor in `Location.cpp`, and
- created examples of `Location`s in a test suite in `test_Location.u` .

You can verify that you didn't make any typoes by compiling these and building a test program, by typing

```
compile Location.cpp test_Location.u
```

If there are problems, read the error messages and then try to fix what's wrong.

Note that if it compiles successfully, you can run `./a.out`, but at the moment it will just say

```
OK (0 tests)
```

because although we've created examples, we haven't actually got any tests to run yet!

Designing methods

Similar to the data design recipe, we can lay out a method design recipe that will take us through a checklist of steps so we don't skip or forget anything.

1. Describe the method.

   As with the data design process, write out a short description of what the method is to do. Use good verbs, and make good use of the words "this" (referring to the current object) and "given" (referring to parameters). Continuing the fraction example from earlier, one fraction method might be `reduce`, which "*modifies* the numerator and denominator of *this* fraction to be an equal fraction represented in lowest terms." Or `plus`, which "*computes and returns* the sum of *this* fraction and a *given other* fraction."

2. Declare the method (write its prototype/signature) and define a stub.

   Based on the description, you should be able to write out the method's declaration in the `.h` file: this sets all of the parameter types and the return type for the method. This is also a good time to decide whether the method should be an accessor (`const`) or a mutator. (In the fraction example, `plus` would be an accessor, but `reduce` wouldn't.)

   In the `.cpp` file, define the method to do nothing but immediately return a dummy value like `0` or `""` or `false` (or nothing at all if the method returns `void`). This is a "stub" method.

3. Write test cases.

   Write out method calls on existing examples that you added to the test fixture earlier, and also write out what they are *expected* to return. (If none of the values in the fixture are suitable to test what you need to test, you can add more!)

   It is not accidental that this step comes before defining the method; especially as the methods get more complicated, you will often be able to state what the answer ought to be even before you figure out how you will compute it. For example, $\frac{1}{2} + \frac{1}{6} = \frac{2}{3}$, but it might take a moment to figure out how to do that computationally.

4. Compile and run tests.

   At this point, you've written enough to make the compiler happy, so you *can* compile, but because all you wrote was a stub, it won't pass

the test. The test case thus simultaneously serves as documentation of what the method should do, and a reminder that you've not finished writing the method yet. At this point in the recipe you can take a break or work on a different method, and know that you won't forget to come back to it later.

5. Step back, take inventory.

   Look at the methods already defined for this class, the private instance variables you have access to, and any of *those* values' public methods. Make sure you understand what data you can access and what already-written methods you can call to help you solve the problem. (This step becomes more important as the methods get more complex.) For the fractions, the fact that `reduce` is available might help you in writing `plus`!

6. Define the method.

   Improve the definition of the method in the `.cpp` file from its original stub into something that can pass the tests for that method.

7. Test!

   Finally, compile the class and its test suite, and run it to make sure the method you've written does what it's supposed to.

Trying it with a `Location` method

This section will walk you through the process on a simple accessor method, not because the full process is *really* needed for that task but to show you how the recipe works. Specifically, you'll write a method to provide public read-only access to the (otherwise private) first coordinate of a location.

1. Describe the method.

The best verb in this case is simply "returns", because the method doesn't have to compute, build, find, modify, or do anything else complicated. The class in question is `Location`, so the description should use the phrase "this `Location`" to refer to the current object. Fill in the blank in the following description as appropriate:

Returns the _____ of this `Location`.

In the future, your method descriptions will get longer and more complex, but they'll pretty much always include the word "this" (to refer to the current object being processed), and if they require any additional information, you'll use a word like "given" to indicate that.

2. Declare the method (write its prototype/signature) and define a stub.

If we have a good method description, we can now answer a few questions about this method. Actually answer these, based on the description from step 1 above:

What type of value will it be returning, if any? _____

Does it have any "given" values, and if so, what types? _____

Will it modify "this" value? _____

The first question tells us the return type; the second tells us about the parameters; and the third tells us whether this method will be `const` or not. (If the method won't modify "this" value, it is a `const` method.)

In this case, looking at the method description, we know it's returning a coordinate that we know to be an `int`. The word "given" does not appear, so it will not take any parameters. And since the action ("returns") doesn't indicate any modification, this method can be declared `const`. So, edit the `Location.h` file, and in the public section of the class, type in a method header that looks something like:

```
int getX() const;
```

(except call it `getRow` if that's more appropriate). Precede it with a `/* comment */` that includes the method description from step 1.

Then, in `Location.cpp`, write a stub definition for the method. As with the constructor, you'll need to precede the name with a scoping construct so the compiler knows which class it belongs to. It will look something like this (again, substitution `getRow` if that's more appropriate):

```
int Location::getX() const
{
  return 0;
}
```

3. Write test cases.

This isn't a super-complicated method, so one test with two test cases should more than suffice. Edit the `test_Location.u` file, and after the `tests:` line, add a test block as follows:

```
test getX
{
}
```

Inside that block, you'll add test cases that say what expression to check, and what you expect it to return. If you had an example location named `exampleLoc` whose x coordinate was 3, you would write

```
check (exampleLoc.getX())  expect == 3;
```

Write at least two lines that follow that basic format, adjusting them to make use of the correct method names and to use the examples you typed in earlier.

4. Compile and run tests.

As before, you'll compile by typing

```
compile Location.cpp test_Location.u
```

and hitting enter. (In fact, if you're in the same window you typed that before, you can hit a few times so you don't have to keep retyping that.)

When you run it, now, one test fails (because that method is just a stub!), and the resulting message should helpfully tell you both what is expected and what is actually being returned.

5. Step back, take inventory.

We'll talk more about this in class later (and when we have more complex data and methods to work with), but at this point we could note that a method of `Location` has access to its `x` and `y` variables. (Or, its `row` and `col` variables, as appropriate.)

6. Define the method.

Edit the `Location.cpp` file and finish defining the method. The body of the method is written just like the body of any other function, except that it has access to all the member variables and methods of "this" `Location`.

In this case, change the return value of the method to `x` or `row` as appropriate.

7. Test!

Now that you've fully implemented the method, when you compile and run the tests, the test that formerly failed should now pass. (If not, you've got more work to do....)

Some more methods to write

All these are methods appropriate to `Location` that will be useful at some point in writing our maze solver. Follow the design process for each one (although I've already given you reasonable descriptions for three of them).

- `getY` (or equivalent)
- `isEqualTo` determines whether this `Location` is the same as a given other `Location`.
- `toString` makes a `string` with the text representation of this `Location` in the form "`(2,7)`".
- `east` computes and constructs the Location immediately to the east of this `Location`. (Note the verbs used here: it does *not* modify this `Location`!)

Handing in

This week there are quite a few files to hand in. You should probably put all of this into its own directory; if it were called `location_lab` you would go to the directory that contains `location_lab` and type

```
handin cmsc162 lab2 location_lab
```

Of course, if you called the directory something other than `location_lab`, type that instead!

This lab is due at 4pm next Wednesday. I expect you'll be mostly done by the time our class meets, but this later time gives you a chance to make edits if you have any last-minute questions.

Rubric (tentative)

RUBRIC

General
1 Present in lab
1 Readme with required stuff
Data recipe (`Location`)
1 Description in comment preceding class definition
1 Examples of `Location` values in test file ♣
1 Full class definition (including constructor, but not other methods) ♣
Methods
1 Header, test, implementation for both `get` methods ♣
1 Correct header, good test cases, valid stub for `isEqualTo` ♣
1 Correct header, good test cases, valid stub for `toString` ♣
1 Correct header, good test cases, valid stub for `east` ♣
1 Full correct method definitions ♣

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