Lab 2

30/31 August 2021

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.

To start off the lab period, I'll begin most labs with a "feature of the day" involving Vim or the command-line interface or both, to help you become a power user of the system. Try not to spend more than 5–10 minutes on it (because there's other stuff to do!), but remember them and refer back to them until the commands they teach are part of your muscle memory!

Vim FOTD: Cut and paste

Open a new file called `dummy.txt` in vim. Enter insert mode (by pressing '`i`'), type two lines of text, and then escape back to command mode.

Somewhere in the middle of the first line, press '`x`' a few times. (This removes characters.) Now press '`p`' a few times.

Somewhere in the middle of the second line, press '`x`' again. Now press '`P`', that is, Shift-P—notice what it did differently?

Go back to the first line, and type '`dd`' (i.e. press the '`d`' key twice). Press '`p`' a few times.

Go to the new first line, and type '`dd`' again. This time, press '`P`' (Shift-P) a few times.

What's happening here is that every time you use a vim command to delete something,[1] it's stored in a clipboard (as if you'd selected "Cut" in a GUI word processor). Then, you can use one of the two paste commands to put the text back—where it was, somewhere else, or any number of times.

There are two kinds of cut commands: those that remove some number of characters and those that remove some number of lines. You've now seen one of each ('`x`' deletes the character under the cursor, and '`dd`' deletes the line under the cursor). The paste commands differ in that '`p`' means "paste clipboard contents *after* this spot" while '`P`' means "paste clipboard contents

[1] NB: this doesn't include using the Backspace key while in Insert Mode.

before this spot". If the contents of the clipboard were character-based (like if you hit '`x`'), then "this spot" means "the character under the cursor", and if the contents were line-based (like if you hit '`dd`'), then "this spot" means "the line under the cursor".

A few more occasionally-useful delete commands:

`dw`	Delete to end of current "word" (char-based)
`db`	Delete back to beginning of current "word" (char-based)
`d$`	Delete to end of current line (char-based)
`d}`	Delete to next blank line (line-based)
`7dd`	Delete seven lines (line-based) (similarly for other numbers)
`dG`	Delete to end of file (line-based)

For each of these, replacing the `d` with a `y` makes Vim do a copy instead of a cut. (The mnemonic for `y` is "yank".)

Beginning of stuff that you read in the preview ←

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, which is meant to represent Cartesian coordinates, points on a (discrete) Cartesian grid. Its purpose will (eventually) be to represent locations of the start and finish in a rectangular-grid-based maze, as well as other locations both inside and 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 on the worksheet 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?

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). On the worksheet, write down at least a few specific examples of `Location` data. Name each example. Make notes about why they are significant and/or interesting.

New stuff starting here! $\leftarrow$

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
{
  private:

  public:

};
```

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 we discussed in class, 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 the *declaration* of the constructor, that is, its header followed by a semicolon. In the ongoing fraction example, this line would probably look like

```
Fraction (int num, int denom);
```

so for `Location` the equivalent would be

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

(For the rest of this handout, I will typically show examples based on the fraction case study, and ask you to translate that yourself into what you need to do for `Location`.) 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. The header in the *definition*, in the `.cpp` file, is almost the same as the header we wrote in the `.h` file, but since it is not directly enclosed by the class declaration, we need to indicate its scope by preceding the method name with the scoping operator `::` and the name of the class. In the fraction example, that much would look like this:

```
Fraction::Fraction (int num, int denom)
```

To actually put the values into the instance variables, we have two options. In class we used one, making the assignments in the body of the constructor method; here I'll demonstrate the use of *initializers*, which are more clearly and explicitly providing the *initial* value of each instance variable in a comma-separated list before the constructor body (which then won't have to do anything at all). The fraction example would look like this:

```
  Fraction::Fraction (int num, int denom) : numerator{num}, denominator{denom}
  {
  }
```

What will the analogous implementation be for `Location`?

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, an example of a fraction, called `threeFourths` and representing $\frac{3}{4}$, would be encoded as

```
    Fraction threeFourths = Fraction{3,4};
```

Go ahead and encode your `Location` examples into the `fixture` section of this file. 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!

Beginning of more stuff that you read in the preview ←

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.

   Declare: 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.)

   Stub: 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. On the worksheet, 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. On the worksheet, answer these questions, 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, on the worksheet, write out a method header. In the fractions example, you'd be writing something like

```
int getNumerator() const;
```

What would be an appropriate name and header for the class we're writing now?

Once you are typing these in (and, in the future, you can type them in directly and not write them by hand on a worksheet, of course), you'll put this in the `Location.h` file, and precede it with a `/** comment */` that includes the method description from step 1.

Next, on the worksheet (and eventually in `Location.cpp`), write out a stub definition for the method. The method header here in the definition is (as usual) nearly the same as in the method declaration (without the semicolon at the end); but for *methods* the name is preceded by a scoping operator to indicate what class it is part of. It will look something like this:

```
int Location::______________________ () const
                  NAME OF METHOD
{
    return 0;
}
```

3. Write test cases.

This isn't a super-complicated method, so one test with two test cases should more than suffice. As you saw in the previous lab if you read the tests I provided, the Unci format has a fairly straightforward way to encode a test plan; every expectation is written in the form

`check (` $\underbrace{\rule{9em}{0.4pt}}_{\text{Expression to evaluate}}$ `) expect` $\underbrace{\rule{5em}{0.4pt}}_{\text{Expected result}}$ `;`

For example, in the fraction example, if you had an example named `threeFourths` representing $\frac{3}{4}$, you might write

```
check (threeFourths.getNumerator())  expect == 3;
```

Expectations can take a few forms, including:

```
... expect == 3; // or some other exact value
... expect about 3.0 +- 0.001; // if the result is inexact
... expect true;
... expect false;
```

Right now, on the worksheet, write out two expressions to evaluate that make use of the method we're designing, and their expected results. You can and should make use of the examples from the data design (that's why we named them); and remember that you can add more!

New stuff starting here! ←

Edit the `test_Location.u` file, and after the `tests:` line, add a test block. Its name can be arbitrary but should generally correspond to the method it's testing. The corresponding block for the fraction example would be

```
test getNumeratorSimple
{
  check (threeFourths.getNumerator()) expect == 3;
}
```

In your version, include the `check-expect` statements (at least two of them) that you wrote out on the worksheet.

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 instance variables (representing the actual coordinates).

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 the name of your first instance variable.

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....)

One last file to add

Going forward, every time you hand in a program or lab work electronically, I will expect the submission to include easily-found documentation, probably in the form of a file named `README.txt`. Such a file should contain:

- Your name
- What lab (or exam or whatever) it is

There will be just one file like this in each submission, but there might be multiple runnable things as part of the task; for each runnable thing, you should have instructions on

- How to compile it
- How to test it (if this is appropriate)
- How to run it/use it (if this is appropriate)

Ideally, these instructions are set up so that the actual literal exact thing to type at the command line is on a line by itself (or lines, if it takes multiple commands). For instance:

```
To compile and run the test cases:
  compile Location.cpp test_Location.u
  ./a.out
```

By isolating these things on their own lines, it is really easy to triple-click them (to select and copy them) and then paste them into a window with an open command line. (See my readme from Lab 1 for some examples of this format.)

These files are human-oriented documentation and can be somewhat free-form, and are a good place to put any other information you think I need to know about your work (or even notes to your future self for if you ever look at this code again). You can vary the format a little bit when that suits the content, but I always want a readme and I always want it to contain at least this stuff. Add a readme file to your directory now, before writing the rest of the methods.

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 (I've already given you the descriptions).

- An accessor for the other coordinate
- `toString` makes a `string` with the text representation of this `Location` in the form "`(2,7)`".
- `isEqualTo` determines whether this `Location` is the same as a given other `Location`.
- `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, but I want a single handin with all of them. 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! And if you run it *from inside* the directory with all the files in it, you can just type

```
  handin cmsc162 lab2 .
```

(don't forget the dot at the end). (You can resubmit as many times as you want, but each submission should have *all* the files required.)

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

Rubric (tentative)

RUBRIC

General	
1	Preview stuff done before 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 at least constructor) ♣
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).