Lab 2
Preview

28 August 2023

This week's lab is all about class design. Before you come to lab, you should read about the design process (ie this PDF) and do some of the early design steps for the class you'll be writing in the lab by writing out your design notes on the worksheet I've provided (which you need to have with you during lab). (You can recreate the worksheet in your notebook if you lose it and don't want to print it out again.)

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.

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.

The rest of the data design steps, as performed for `Location`, will be done in lab.

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 ( _____ ) expect _____ ;
        EXPRESSION TO EVALUATE   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!

The rest of the method design steps, as performed for this method, will be done in lab. Now, finish the last part of the worksheet by performing these steps for the `isEqualTo` method of `Location` as well.