Lab 4

12 September 2023

Today you'll experiment with some code to start to understand how C++ pointers work. But first:

Vim FOTD: "ex" mode

Open two terminal windows, and arrange them side-by-side. The one on the left I will designate the "edit" window, and the one on the right the "other" window. In the edit window, edit a file named `dummy.txt`, type a few lines into it, and save and quit. (The content doesn't matter at all.)

In the other window, `cat` the file, that is, type

```
cat dummy.txt
```

You should see the exact contents you just typed in. If not, make sure both windows are in the same directory (your home directory is fine) and try again.

Return to the edit window, again edit the `dummy.txt` file, and add a couple more lines. This time, don't save and quit. Back in the other window, `cat` the file; since you haven't saved yet, what should you see?

Return to the edit window again. Make sure that you are in command mode (hit escape), and this time, instead of save-and-quit (with `:wq`), just save: if you type `:w` by itself and hit enter, this writes the file without quitting. Now in the other window, `cat` the file again—you should now see the updated version.

What you are seeing here is (more of) a third mode of Vim besides insert and command mode, called "Ex mode". Ex was an editor back in the day, a precursor to vi (which was itself the basis for vim). All interactions with the ex editor were done through commands typed on a line that started with a colon, and vestiges of this survive in the modern Vim editor; from ex mode you control file access operations, among other things. The `:w` command that you have just seen simply saves ("writes") the current file.

Another command is to save to a different file. After you issued the `:w`

command a moment ago you were returned to command mode, so press the colon key again to return to ex mode and type

```
w dummy2.txt
```

This creates a brand new file, named `dummy2.txt`, with the current contents of the edit buffer. But unlike "Save As..." in a typical modern word processor, it doesn't change the default name of the file, so that if you type `:w` again, it will save it under the original name (`dummy.txt`).

You can use ex mode to edit a different file. Type

```
:e newfile.txt
```

and the `dummy.txt` file will disappear from the window, to be replaced by an empty buffer. If you type text in here and then hit `:w`, it will be saved under the name `newfile.txt` (as you can verify by using `ls` and `cat` to look at the file from the other window).

If you then type

```
:r dummy.txt
```

it will read the full contents of `dummy.txt` into the current buffer.

Ex mode can be used to quit Vim by typing `:q` and hitting enter. Note that this does *not* include writing out the file first. If you type `:q` before saving (and you can tell that the file has been edited by the "`[+]`" after the filename in the status bar), Vim will warn you that you haven't saved the file. You can then type

```
:q!
```

to say, no really, I mean it, just quit (don't save).

There are other ex mode commands that we'll learn eventually, but these file-related commands enable a particularly useful interaction style: you can now leave your source code open in one window, save it, and run the compiler in the other window. This is useful so that you can keep any compiler errors on the screen while you scan the code for the problem; in fact, from now on you should get in the habit of having at least two windows open when you're programming: one for editing, and one for compiling and testing. (Some of you have already been doing this, but it's even more streamlined now.)

Cards to play with

We've been using the example of a playing card class, and in the lab we'll continue with it. To get us started, I've put an upgraded version of that class in `/home/shared/162/lab4/` to save you some typing. Copy those files into the directory you create for this lab.

Look at the files, but don't modify them. (From vim, if you haven't changed anything and type `:q`, this quits without saving. If you accidentally make changes inside vim, and still want to exit without saving, you can type `:q!` to indicate that you really mean to quit without saving.) Read the parts of the code that are new, and make notes of anything you want to ask about (there will be time for this at the start of lab, and there are definitely at least a couple techniques and C++ features that are probably new to you).

Constructing

Create a new file that you will call `cardmain.cpp`, and set it up to have a `main` function in the usual way (with `#include` lines and so on). In addition, be sure to `#include <memory>`. Type this in as the body of `main`:

```
Card* a = new Card (7, Card::SPADE);
shared_ptr<Card> b = make_shared<Card> (7, Card::SPADE);

cout << a->toString() << endl;
cout << b->toString() << endl;
```

Compile the program (remember that you'll need to compile it together with `Card.cpp`) and run it. You should see

```
7S
7S
```

If not, check carefully that you typed what I wrote above, and do your best to remove the error.

Now, look back at those lines above. Identify the three places that this code uses a syntactic feature we've not really seen before this week's readings, and in your notebook, write down what each of them means or does. (You'll probably want to refer back to your book or notes for this.)

Equality

Add the following lines to the program:

```
shared_ptr<Card> c = make_shared<Card> (7, Card::SPADE);
shared_ptr<Card> d = make_shared<Card> (4, Card::HEART);
shared_ptr<Card> e = d;

cout << b->toString() << c->toString() << " b==c? " << (b == c) << endl;
cout << d->toString() << e->toString() << " d==e? " << (d == e) << endl;
```

Compile it, and run it. These two additional print statements do not produce the same result. Why not? Continue on the same notebook page as your earlier work and give your explanation, including a diagram of memory after all these variables have been set.

Dereferencing

In some circumstances, we will have a pointer but need to make use of the value it points to. First, let's see what happens if we have a mismatch—add the following lines:

```
cout << "b eq c: " << b->isEqualTo(c) << endl;
cout << "d eq e: " << d->isEqualTo(e) << endl;
```

Compile it and read the error message: in your notebook, write out the most salient phrase(s) from the (several lines long) error message.

Fix the two lines by adding asterisks to the method parameters:

```
cout << "b eq c: " << b->isEqualTo(*c) << endl;
cout << "d eq e: " << d->isEqualTo(*e) << endl;
```

Compile it and run it. How does this output compare to the earlier lines? (Write your answer in the notebook.)

Updating the dereferenced value

With pointers, you can change the value they point to (without "moving the arrow"). That means that anything else that points to that spot will also effectively be updated. Try this:

```
*b = Card{9, Card::CLUB};
cout << b->getRank() << b->getSuit() << endl;
cout << c->getRank() << c->getSuit() << endl;

*d = Card{2, Card::DIAMOND};
cout << d->getRank() << d->getSuit() << endl;
cout << e->getRank() << e->getSuit() << endl;
```

Null pointers and language versions

Add the following code to the program:

```
shared_ptr<Card> f = nullptr;
```

Compile it. Now is a good time to remind you that we're using the C++17 standard language, i.e. the version codified in 2017, and a lot of what we're doing here is newish as of C++11, and a *lot* of code out there is still written against C++03 or earlier standards. `nullptr` is a more typesafe version of what used to be written as `NULL`.

Pointer errors

Pointers get something of a bad rap because they can be the source of some subtle (and not-subtle) errors in your code. In this section of the lab, I am having you type in some intentionally-incorrect pointer code so that you can see the error it generates. For each of the following pieces of code (most are one line, some are two), add it at the end of the program and compile it (and run it, if the compile was successful). Then, write down what the error message was, if any (just a phrase or two if the error is long), and what was actually wrong with the code. Then, delete that erroneous piece of code before trying the next one.

```
cout << a.getRank() << endl;
```

```
cout << b.getRank() << endl;
```

```
cout << f->getRank() << endl;



cout << b << endl;



Card* g = b.get();
cout << g->getRank() << endl;
delete g;
```

Finishing up

If you get this far before the end of the lab period, compare notes with the other students in the lab, and help them if they're stuck on some of the error messages or other things they're supposed to write in the lab. (That doesn't mean give them your work to copy, but do give assistance.) If you've come to different conclusions about some part of the lab, try to resolve it, but if not, that would be a great thing to ask about in class tomorrow!

Handing in

This isn't a project-oriented lab, so there's no code to hand in, but your work will lead into class discussion tomorrow, and from there into a homework assignment (which will be done on paper). So, no need to run `handin` this week. (At least, not for this lab; the handin for Lab 3 is due on Monday, and don't forget to keep poking at the Lab 1 problems if you've gotten less than ten done.)