Lab 4

7 February 2019

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

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 *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:

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

`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".)

Cards to play with

I often use the example of a playing card struct or class, capable of representing the rank and suit of a card. I've put a *very* simple 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.)

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, 'S');
shared_ptr<Card> b = make_shared<Card> (7, 'S');

cout << a->getRank() << a->getSuit() << endl;
cout << b->getRank() << b->getSuit() << 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 Wednesday's class and reading, 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, 'S');
shared_ptr<Card> d = make_shared<Card> (4, 'H');
shared_ptr<Card> e = d;

cout << "b==c: " << (b == c) << endl;
cout << "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, 'C'};
cout << b->getRank() << b->getSuit() << endl;
cout << c->getRank() << c->getSuit() << endl;

*d = Card{2, 'D'};
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 point out that we're using the C++11 standard language, i.e. the version codified in 2011. (Previous versions include C++98 and C++03, and more recent versions include C++14, C++17, and the still-in-development C++20.)

A lot of the things that we are using are features that have been in C++ for decades, but others were new in C++11. For instance, `nullptr` is a newer way of writing what used to be called `NULL`. The newer form helps the compiler write better error messages, but is otherwise basically equivalent to the old one in its semantics. All of the smart pointer stuff (`shared_ptr` and so on) are C++11 developments.

These days, C++11 is often the default language version for a compiler, but even now there's a lot of older code out there. If you look at other books, or read tutorials online, or work with other people's code, you should be aware that the language changes over time.

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.