Lab 4

18 September 2014

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

Vim FOTD: cut, paste, and movement

Many of you already know that you can type d d in command mode to delete the current line. Some of you also know that after you've done that, you can go someplace else in the file and press p, and it will paste it in again, right below the cursor.

Open up a file and try it! Notice that you can paste as many times as you want (just like with cut and paste). Also try Shift ⇑ P, which pastes the line *above* the cursor.

A related command is "yank", which you would know better as "copy": if you press y y, it stores the current line for pasting without removing it first.

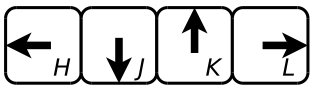
A third command is "change": if you press c c, it removes the current line and puts you in insert mode to type stuff there; after you press Escape to return to command mode, the line that was removed is ready for pasting with `p` or `P`.

But all three of these commands are much more powerful than just working on a single line. If you precede them with a number, it works on multiple lines (so, 1 3 y y yanks 13 lines—try it!). And all of them work together with a variety of movement commands. Try some of the following commands (all from command mode, of course):

`dG`	Delete to EOF	`yG`	Yank to EOF	`cG`	Change to EOF
`dw`	Delete word	`yw`	Yank word	`cw`	Change word
`d}`	Delete to blank line	`y}`	Yank to blank line	`c}`	Change to blank line
`d$`	Delete to end of line	`y$`	Yank to end of line	`c$`	Change to end of line

These are all special cases of a rule that pairs the three commands (`d`, `c`, and `y`) with movement commands: you can delete, change, or yank from the current position to anyplace you can move to using Vim movement commands, which include `G` ("go to end of file"), `w` ("go to beginning of the next

word"), and many others. This table gives a (non-exhaustive!) list of several of them—try some of them out!



Key(s)	Movement
`h`	Left one character
`j`	Down one line
`k`	Up one line
`l`	Right one character
`^`	To beginning of current line
`$`	To end of current line
`Ctrl-F`	Forward one page (screen)
`Ctrl-B`	Back one page (screen)
`G`	To last line of file
`#G`	To line # (e.g. `2G` to go to second line of file)
`w`	To beginning of next punctuation-delimited "word"
`W`	To beginning of next whitespace-delimited "word"
`e`	To end of this punctuation-delimited "word"
`E`	To end of this whitespace-delimited "word"
`b`	To beginning of this punctuation-delimited "word"
`B`	To beginning of this whitespace-delimited "word"
`}`	To next (batch of) blank line(s)
`{`	To previous (batch of) blank line(s)
`%`	To matching paren/bracket
`[(`	To previous unmatched left paren
`[m`	To start of current function

Some of these are more mnemonic than others, of course. The first four are not mnemonic at all, but super-convenient once you've got them in muscle memory, because they're right in the home row, so your fingers don't have to go anywhere to type them.

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 your directory 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

Open another 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). Type this in as the body of `main`:

```
Card* a = new Card (7, 'S');

cout << a->getRank() << endl;
cout << a->getSuit() << endl;
```

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

```
7
S
```

If not, check carefully that you typed what I wrote above, and if you're still having trouble (it doesn't compile, or it produces the wrong result), call me over to help fix it before you get too much farther.

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

1.

2.

3.

Equality

Add the following lines to the program:

```
Card* b = new Card (7, 'S');
Card* c = new Card (4, 'H');
Card* d = c;

cout << "a==b: " << (a == b) << endl;
cout << "c==d: " << (c == d) << endl;
```

Compile it, and run it. These two additional print statements do not produce the same result. Why not? Give your explanation below, 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 << "a eq b: " << a->isEqualTo(b) << endl;
cout << "c eq d: " << c->isEqualTo(d) << endl;
```

Compile it and read the error message: in the space below, 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 << "a eq b: " << a->isEqualTo(*b) << endl;
cout << "c eq d: " << c->isEqualTo(*d) << endl;
```

Compile it and run it. How does this output compare to the earlier lines?

Null pointers and language versions

Add the following code to the program:

```
Card *e = nullptr;
```

Compile it. Now is a good time to point out that when you run "`compile`", it is running a slightly more complicated command for you, which you can see it print out before proclaiming "Success!". One piece of that is the option "`--std=c++11`", which says 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 the latest version C++14 was just approved a month ago. If not specified, most current compilers default to C++03.)

A lot of the things that we are using are features that have been in C++ for many years, but a few are new. For instance, `nullptr` is a newer way of writing what used to be called `NULL`, and 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. Also, when (eventually) we move away from always using the `compile` command, you'll have to remember to specify C++11 on your own, or code that uses things like `nullptr` may give you error messages.

For now, though, `compile` will take care of all of that for us. Moving on!

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 << e->getRank() << endl;



cout << a << endl;



delete c;
delete d;



delete c;
cout << c->getRank() << endl;
```

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.