# Lab 5
## Loop practice

*29 September 2015*

## Loop drill

Lab this week is getting more practice with the loops you read about in §4.4.2. The "drill" is essentially to get started with the first couple short practice problems.

For each of these, write a program whose `main` contains the required thing. Keep track of which file is which in your readme, and dont forget to test that they work correctly! (These in the drill dont even *have* input, so testing them will be pretty easy!)

a. The `while` loop on p109, except stop at 9 instead of 99. Where it says "`square(i)`", use "`i*i`" instead.

b. The `for` loop on p112, except stop at 9 instead of 99. Again, use "`i*i`" to compute the square of `i`. (Note that the output for this should be identical to the first one.)

c. A `for` loop that prints out the phrase "Aloha, world!" 20 times.

d. A loop of either type that solves the "Try this" on p111 (involving printing a character table).

I'll be circulating around the lab to answer questions. If you're stuck on some part of the drill, ask me about that (and while you're waiting for me to get to you, look at the next sections about cut and paste, and diff). If you're not stuck but haven't finished the drill, work on that now. If you're done with the drill, continue on to the next section.

## Vim FOTD: Cut and paste

Open a new file called `dummy` 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:

| | |
|---|---|
| `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”.)

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

## Command line FOTD: advanced `diff`

Up until now, when you've used `diff` to help you test your code, it's been a bit clunky, because it's two steps. First, you send your output to a file:

```
  ./a.out  <test1.in  >test1.out
```

and then compare it to what it was supposed to be:

```
  diff test1.expect test1.out
```

It gets even more unwieldy with multiple test cases (as you discovered in the last lab!). Now that you're doing this sort of thing a lot more, it will be more convenient to combine those two steps into a one-liner, of the following form:

```
  ./a.out <test1.in  | diff test1.expect -
```

What has changed? It starts out like the first line from before (run `a.out` with input from `test1.in`), but then instead of sending its output to a file, it uses that vertical-bar symbol (shift-backslash) to send it directly to the `diff` program! The `diff` call is, in turn, modified to indicate that the "actual" input is being received directly rather than from a file (indicated by writing a hyphen where you previously wrote a filename).

Try it now! Go into a previous lab directory where you have a program and its test cases. Type the above one-liner version, replacing `test1.in` and `test1.expect` with the names of your actual test case files.

## More loop practice

For the rest of the lab period, work on writing programs that accomplish the following tasks. Don't forget to test that they work correctly. For the string-related ones, remember that Lab 3 listed a variety of useful string operators and functions.

e. Reads a single positive integer and then prints a countdown: The first time through the loop, prints "99 cans of White-Out Mountain Dew on the wall..." (or whatever the input number was), then "98 cans of White-Out Mountain Dew on the wall...", counting down to 1. Feel free to substitute different contents for the cans. (Hint: for your test case, don't use 99!)

f. A loop that keeps reading numbers and adding them to a total, stopping only when the total becomes greater than 21. For example, one time it might read 10, then 10, then 2, and stop; another time, it could read 4, then 5, then 6, then 6, then 9, and stop. After it stops, it prints the number of inputs it read, and what their total ended up being.

g. Reads a single "word", and examines each character of the input one-by-one and prints them back out unchanged except that if it's an underscore, your program prints a space instead.

h. Reads in ten "words" (as delimited by whitespace), and prints them back out each separated by a space EXCEPT that if one word is the string "`/*`", stop printing out words until you read a "`*/`". For instance, if the input were

```
testing the output /* and what shouldn't print */ today
```

the expected output would be

```
testing the output today
```

This is a bit like what goes on in the compiler when it ignores your comments.

i. Reads in two positive integers (the first less than the second) and prints out all the even numbers between them (including the numbers themselves, if they're even). So if the inputs were 6 and 10, the program would print 6, 8, and 10. (Hint: what's the remainder when you divide an even number by 2?)

## Handin (tomorrow!)

I don't need you to run the `handin` command this week, but I do want you to bring your work to class tomorrow. This is easiest if you just bring your laptop to class with at least an hour of battery or a power cord (assuming it can access the campus wireless network); if not, copy down what you have into your notebook (or other paper) before coming to class.

It's ok if you're not 100% done with all the problems, or if you're stuck on some of them, but you should make an honest try at each of them. If you're not sure how to implement them in C++, try implementing *part* of them, or writing pseudocode. Do your best!