Lab 7
Divisors

20 October 2015

The drill for this lab is the **Chapter 5 drill**. Come to lab on Tuesday either with it completed or with a specific written question in your notebook identifying which drill step you got to and what about it you're stuck on.

Given the format of the drill, if you do get stuck on how to fix one of them, move on to the next one! Since the 25 elements of the drill can all be typed in independently of each other, there's nothing preventing you from typing them *all* in, even if some of them aren't fixed yet.

Notes and adjustments:

- When you type in the scaffolding, omit the lines that call `keep_window_open();`.
- For each of these, you'll be saying, in your readme, what was wrong, *not just* how you fixed it. For instance (freebie!) on the first one, it's not enough to just say "replace Cout with cout", or "change C to c"; you should also say, "C in Cout is uppercase, so it's not seen as the same thing as cout". Or something like that.
- In numbers 13, 14, 15, and 19, insert the keyword `unsigned` before the keyword `int`. The absence of the `unsigned` is not the problem, if any, with those lines.
- The programs should compile, not just without error, but without any warnings either (once you fixed the `unsigned` issue I mentioned above).
- But to fix a program, it is not sufficient to just get it to compile and run without throwing an exception; it needs to print, exactly, the string "`Success!`" followed by a newline. (If it isn't followed by a newline, your prompt will get printed on the same line as the "`Success!`", which looks funny and is incorrect.)
- Below is a list of "par" values for each of the 25 problems. As in golf, you want to try to find a fix with the lowest number of strokes—in this case keystrokes. The number represents, roughly, the number of

characters you can add, delete, or modify in order to fix the line. In a few cases, you might do it with an even smaller number than I've given. If you go over by one or two, that's probably fine. If you go over by a lot, give it a closer look—you might not really be understanding the problem(s) with the line. (But, if you don't come up with anything better, still put that in as your answer! There's partial credit to be had here.)

Prob	Par	Prob	Par	Prob	Par	Prob	Par
1	1	8	5	14	2	20	1
2	1	9	2	15	1	21	2
3	2	10	1	16	4	22	6
4	2	11	2	17	4	23	2
5	5	12	2	18	2	24	2
6	5	13	1	19	5	25	3
7	4						

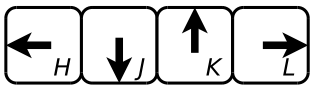
- Don't forget that some lines may have multiple errors, and some lines may have none. (There's at least one that has none, but I won't tell you which one(s) or how many. I've given it/them nonzero par value(s) in the table above just to keep things fun.)
- When you type them in, it's ok to spread the given code onto multiple lines rather than typing them as one-liners as in the book. (This is especially helpful for the later ones that have several statements and control structures.)

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 section about vim movement commands). If you're not stuck but haven't finished the drill, work on that now. If you're done with the drill, go on to the next section.

Vim FOTD: movement keys

Vim responds to the arrow keys and keys like PageUp and PageDown, but there are a number of additional keys that can be pressed in command mode to move around the file. Open one of the files you have lying around and 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. 1G to go to top of file or 23G to go to line 23)
`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.

So what, right? Well, all of the delete commands that you learned in earlier labs were special cases of a rule: `d` plus a movement command deletes from "here" to wherever that movement goes. So, `d1G` deletes to the top of the file. And `d%` deletes everything between this paren and the matching one. Since the `p` command only pastes the most recently-deleted thing, it's very helpful to be able to delete everything you want to "cut" all at once. Same goes for the `y` ("yank", i.e. copy) commands. The re-indent command (`=`) is another one that works with an arbitrary movement: `=%` reindents everything between "here" and the matching paren or bracket, while `=G` reindents everything from "here" to the end of the file, and so on.

There's no need to memorise all the movement commands right now, of

course. A couple might stick, but for the rest, even if you don't remember the command, you'll remember it exists, and you can always come back and refer to this sheet.

Reading and debugging

Your other task this week involves reading some code that I wrote. I've placed a copy into a shared directory; copy `lab7.tar` from that directory to your own as follows (assuming you're in your own lab directory):

```
cp /home/shared/160-3/lab7.tar .
```

(don't forget the lonely dot at the end to put it in the current directory). Then, type

```
tar xvf lab7.tar
```

to unpack it.[1]

I've named the readme `README1.TXT` so it won't overwrite your own readme if you've already started one—you'll want to absorb its contents into your own readme, though. (One way to do this is to do it from within vim: in command mode, type

```
:r README1.TXT
```

and it will read the contents of that file into the current buffer.)

Look in particular at the readme, and at the code in `listDivisors.cpp`. The program is structurally more or less correct, but the code is rife with bugs in several categories. Use your debugging skills to find and fix them; and for each bug you fix, classify it and record it in the readme. There are four major categories of bug here—compiler error, linker error, runtime error that terminates the program, and logic error—and this program code contains at least one in each category. Your readme should say for each bug both what the bug was (and how you fixed it) and which of those four categories it belongs in.

[1] The `x` stands for "extract", the `v` is for "verbose", and the `f` is for "file", as opposed to the magnetic data tape that such Tape ARchives were once stored on. `tar` is an old, old program.

In the course of your debugging, you might end up adding comments to the code, which is fine, and you might add some print statements as well, which you may leave in the code but should comment out. You should also edit the program comment at the top of the file to reflect your authorship (add another `@author` line with your name) and any changes to the description as necessary.

Handing in

As usual, use the `handin` program. Designate this as `lab7`. Hand it in by 4pm on Monday, 26 October.

Rubric

RUBRIC

1 Attendance at lab with drill done or question written down

Drill, each

✓ Error found, fixed, identified in readme
✓ Non-error clearly indicated in readme
∗ Error fixed correctly, but omitted or misidentified in readme
∗ Error identified in readme, but unfixed
∗ Error fixed with lots of extra unnecessary work
× Problem skipped or answer not worth credit

Drill, points

1 At least 10 ✓s (**½** for 8 ✓s, or ✓ + ∗ ≥ 10)
1 At least 16 ✓s (**½** for 14 ✓s, or ✓ + ∗ ≥ 16)
1 At least 21 ✓s (**½** for 19 ✓s, or ✓ + ∗ ≥ 21)
1 All 25 ✓s (**½** for 23 ✓s, or ✓ + ∗ ≥ 25)

listDivisors

1 At least 2 errors found, fixed, id'ed in readme
1 Compiles to executable
1 All compiler and linker errors id'ed in readme
½ All but one id'ed; or, all id'ed as C/L but some swap C/L
1 Runs correctly
½ Displays divisors on at least some numbers
1 All runtime crashing and logic errors id'ed in readme