Lab 2
Conditionals

5 September 2019

The drill for this lab is below. Come to lab on Thursday 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.

1. First, open up the repl.it assignment "Lab 2 drill". In the pane on the left, read in a grade (which will be a number, possibly with a decimal, between 0 and 100). (No need to print a prompt here—and if you do it'll mess up my test cases!)
2. Below this, write an `if` statement that will print the letter grade "A" for any grade that would get an A or A− in my course (hint: see p3 of the syllabus). (We're ignoring the +/− for this drill just to make it a bit less tedious. Otherwise we'd need twelve different conditions!)
3. Remember how you're supposed to try running the program after each drill step? If you haven't yet, make sure to do that now (and fix any errors so far).
4. Now use `else if` to print "B" for any grade that would get a B+, B, or B−. Remember that any number that would get an A has already been handled.
5. Do likewise for C and D.
6. Finally, use plain `else` to print "F" for any grade that does not appear higher on the grading scale.
7. Did you remember to put comments at the top of the program? See Lab 1 for a list of what to put there.
8. Then, if all the tests are passing, click submit.

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 dotfiles). 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.

Command line feature of the day: Dotfiles

From your home directory, type

```
ls -a
```

It should list a bunch of files you don't normally see that start with a period. Many command line programs have a number of configuration options that the user can set to control how they appear and how they behave. To make these config files easily accessible, they were put in the user's home directory; and to make them unintrusive, they were named by convention with names beginning with periods, which `ls` doesn't normally list. Such config files are popularly known as "dotfiles".

Let's start by doing some things to configure Vim, since we spend so much time using it. Edit (using vim, of course) the `.vimrc` file. (It probably doesn't exist yet unless I've already helped you set it up.) Add the following lines to it:

```
set number
syntax on
```

You'll be typing configuration options into this file, and then seeing what happens when you edit your C++ code and other files; this will be easiest to see if you open a second terminal window, and in the left one edit the `.vimrc` (in your home directory) and in the right one edit your C++ files (in one of your lab directories).

Right away, when you edit a C++ file you should see a difference: the syntax highlighting and the line numbers that you've seen when I edit code.

Now, in the `.vimrc` file, add the line

```
set laststatus=2
```

and save it. Now, edit a C++ file. Do you see the difference?

(Go ahead, look for it.)

The change is that there is a status bar on the (second-to-) last line of the window which contains the name of the file (useful!) and, once it has been edited, the symbol `[+]`.

Here are a few other things to add to your `.vimrc`:

```
set cindent shiftwidth=2
set showmatch
```

Try adding them, and see if you can figure out how they change vim's behaviour when editing a C++ file. We'll talk about them in class tomorrow, and you can see if you guessed right.

One last one: PuTTY defaults to a terminal window with dark background and white letters; and you may notice that this makes Vim's syntax colouring a little hard to read. If so, you can add

```
set background=dark
```

to your .vimrc to improve the contrast there. (The default colour scheme is designed for light backgrounds; you can ask for it explicitly as "`set background=light`" if you prefer it.)

Another dotfile that you probably already have is a `.bashrc`, which controls how your command-line shell runs. (The name of the shell is "`bash`", which stands for "Bourne-again shell", a very geeky joke whose explanation you can google for yourself.) From your home directory, edit this file.

You might have one already; depending on when your account was created, it might have a lot in it or just a few lines. Some other time you might want to read through it all, but for now I'll just point out one small thing you can edit: your prompt. That's the text that is printed after every command you execute. For now, make it something simple; go to the bottom of the file and insert

```
PS1="Type something: "
```

Inelegant, but we'll fix it in a minute.

Save and quit the editor. Now, the changes haven't taken effect yet, because your current command-line shell was started before you changed the `.bashrc`. One way to check how the changes went is to open a new terminal window—this will make a new shell, using the new settings. Another is to force the `.bashrc` to be reread by typing

```
source .bashrc
```

Not happy with the prompt? Let's change it again. Edit the `.bashrc` file again, and this time change the prompt to

```
PS1="\h \w -\#-$ "
```

(That's approximately what I use.) This gives you, respectively, the name of the machine, the directory you're currently in, and the "command event" number, i.e. a running count of commands you've executed in this shell. You probably don't want to forget the space before the double quote.

After you try that, here's another variation to try (a default on some systems):

```
PS1="[\u@\h \W]$ "
```

Ultimately, your choice of prompt is up to you. Experiment and find something you like!

Back to conditionals: testing

The remainder of this lab is all about thinking about conditional execution, but it takes two major forms, each one as important as the other: one is how to write them in a program, but the other is to think about how to effectively test them. We'll address that part first.

The program you'll be writing—on the department server (via PuTTY, probably)—should perform as follows: it will prompt the user to type three names; it will read the three names; and then it will report whether they are in sorted order or not. You can assume that each name has exactly one capital letter (its first letter) and no spaces or non-alphabetic characters,[1] which means that for any pair of them, the relational operators (such as `<`) will correctly report which order they would be in, alphabetically.

With that much information, you can already write your test cases! In the space on the next page, write out a few test cases. Each test case includes both an input (three names), and the program's corresponding output. (Don't forget that the program will be printing out a prompt first!)

[1] This is an *extremely* unrealistic assumption about names, and you undoubtedly know at least a few people whose names would not work correctly with this program. Stay tuned! We'll talk more about this problem a little later in the semester.

Input	Output

That space above should have *at least two* test cases in it before you move on. (Do you see why two is a clear minimum here?) As you go through the lab, you may discover that there are additional relevant test cases that you aren't handling; always feel free to go back and add more if you think your coverage is insufficient.

Typing in the test cases

First, on the department servers (in PuTTY), create a directory for this lab (probably something like `lab2`). If you don't remember how to create a directory, consult Lab 0 (it's in the summary on the back page). Then change into that directory, so that all your subsequent work will be in there.

Edit a file called `test1.in`, and type in the input half of the first of your test cases above. The names can either be on separate lines or just separated by spaces (either way will not affect how your program runs). Save that and then put the matching expected output into a file `test1.expect`.

Do likewise for your other test case(s), using test2 (and test3, etc, as many as you need).

Writing these files now does a few things. First, you've thought about the problem, its output, and what it means to be "in order" (or not). Second, you've communicated *to me* and to anyone else looking at your directory what you think the program should be doing. That's important! And finally, when you get to the point that you're ready to run the program, even partially, you've got a test system you can automate to let you know what works and what you still need to work on.

Writing the program

This may almost be an anticlimax at this point, but now's the time to edit your program file. The exact name of the file is not important (but it should end in `.cpp`, for "C Plus Plus")—something like `inorder.cpp` would be appropriate here.

Start editing that file using vim. Enter insert mode, and type in the boilerplate stuff that goes at the top of every C++ program. Inside the curly brackets for `main`, type in your prompt, create your variables, and use `cin` to read values for them. If you need to look up how to do any of that, go ahead! You can refer to your drill from this lab (on repl.it), either part of last week's lab, any of the stuff we did in lecture, or the textbook.

Once that's all in (and maybe even before), you can compile and test your program. You've seen before how to compile (consult Lab 0 if you've forgotten), but testing I just showed you briefly in class yesterday. Assuming you've compiled the program, you should be able to type

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

to run a test on it.[2] At this time, the test is guaranteed to report that your program isn't printing the expected output (because your program isn't printing *anything*). That's great! It means your tests are working.

I recommend editing a file called `README.txt`—eventually to contain all sorts of documentation—that, for now, just has in it

```
./a.out  < test1.in  |  diff -sZ test1.expect -
./a.out  < test2.in  |  diff -sZ test2.expect -
```

and maybe more lines if you have more test cases. Then if you type `cat README.txt` at the command line, you can copy-and-paste (in PuTTY, highlight and then right-click) to just run them all.

[2] There's a lot to unpack there, and you don't need to worry about it much (you can just always use lines of exactly this form), but if you're curious: "`< test1.in`" says to use the contents of the file as input instead of getting it from the keyboard. "`| diff`" says to take the results and compute differences. "`-sZ`" are parameters to `diff` that say to say something even when there are no differences (rather than just being silent) and to ignore certain minor whitespace differences. "`test1.expect -`" says the things to compare are the contents of that file, and whatever is printed by `a.out`.

But again, at this point, your tests are not passing because you're not printing anything yet.

Deciding what to print

Your first approach to this should be to break off a smaller piece of the problem: rather than "are they all in order?", you can ask, "what's *one way* I can quickly see they *aren't* in order?" That one way needn't identify all the ways three names can be out of order—just one of them.

For now, you can write an `if` that identifies that one case, and prints that they're out of order (using whatever message you wrote in your expected output files).

Move forward along this line of reasoning: there's more than one way for the names to be out of order. But if none of them are out of order, then they must all be in order. (Hint: it's not a coincidence this falls right after we covered `else if`.)

Remember to test your work. If you think of a different case, a different way the names could be out of order, you can either:

1. write a test case that illustrates it, and then
2. update the program to pass the test case;

or,

1. update the program to work on the new situation, and then
2. write a test case that confirms it works.

I somewhat prefer the first way, but either way will get you full credit. You do want to make sure your test cases catch up with your understanding of the conditional execution, though.

Handing in

It's due as usual on Wednesday at 4pm. Hand in using the `handin` command, this time with assignment name `lab2`. (Check the end of Lab 0 if you don't remember how to run the command.)

Rubric

RUBRIC (Tentative)

1 Attendance at lab with drill done or question written down

Drill (Grades)

½ Documentation at top of file
1 Program runs, prompts, reads values
1 At least one valid `if` comparing input and printing grade
1 Suitable `else if` structure
1 Comparison logic is correct to print correct grades

Checks order

½ Documentation
1 At least one pair of .in/.expect files w/ suitable contents
1 Test case coverage, and TCs pass or are noted in readme*
1 Program compiles, runs, id's one case and responds correctly
1 Program responds to all cases correctly

* To get the point for good test case coverage, you have to EITHER pass all test cases in the group OR indicate in the readme which ones aren't passing. This is your way of showing me that you've actually run your tests. Someone remind me to talk about this in class on Friday.