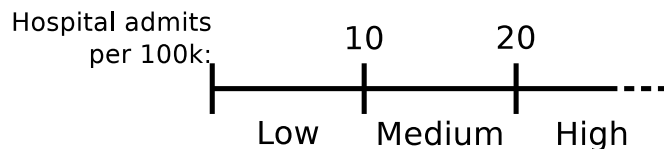


Lab 2

Conditionals

10 September 2024

For this problem, we'll use a slightly abridged version of the standards the CDC developed on how to determine a community's Covid-19 level. If the case rate is otherwise low (below 200 per 100k per week), the community level is driven by how many cases are significant enough to require hospitalisation, on the following scale:



You'll write a program to compute this.

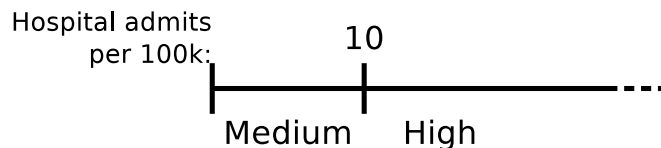
1. First, open up the codeboard assignment “Lab 2 “Covid levels” (pre-view)”. Write code to read in a hospitalisation rate (which will be a number, possibly with a decimal, which you'll interpret as the rate per 100,000 residents in one week—that is, a number on the scale illustrated above). No need to print a prompt here—and if you do it'll mess up my test cases!
2. Try compiling your program if you haven't already. In general, you should compile your program as often as you can, after every couple lines you write. It doesn't *do* anything yet, but if it doesn't even compile, there's only a few places to look at to detect errors.
3. Below the input statements, write an `if` statement that will print “Low” if the hospitalisation rate is below 10.0.
4. Compile it again, and this time try running it—it should print a response in at least some situations. (I won't keep inserting these compile-it-and-run instructions but you should keep doing so, frequently.) If think you've got that much working, click Submit!
5. Now use `else if` to print “Medium” when that classification applies.
6. Finally, use plain `else` to print “High” as appropriate.

7. Did you remember to put your name and a brief description after the `//` at the top of the program?
8. If you think it's correct, or even if you're not totally sure but it's at least compiling and running, click Submit.
9. If any of the cases are failing, try to identify and fix the problem, and then click Submit again.

Continued...

Once you've got the above parts working (submitted and passing all test cases), save the project, select the whole program, copy it (Ctrl-C or Cmd-C), and exit the project and open the project named "Lab 2 "Covid levels" (continuation)". Paste (Ctrl-V or Cmd-V) into the program here: the end of the previous part is the starting point for this part.

It turns out that if the overall case rate is higher (at least 200 per 100k per week), the CDC raises the community level by one step, using this scale instead:



For this continuation, using your existing program as a base, update it so that it reads both the general case rate *and* the hospitalisation rate—in that order, without prompts, or my test cases won't work—and then prints out the community Covid-19 level corresponding to those rates.¹

I'll be circulating around the lab to answer questions. If you're stuck on some part of the preview or its continuation, ask me about that.

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

¹In both versions of the problem I have elided details about rates of filled hospital beds, which were also used in the full CDC formula.

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,² 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 a notebook, make a table in this format and write out a few test cases:

Input	Expected output

Each test case includes both an input (three names), and the program's entire corresponding output. (Don't forget that the program will be printing out a prompt first!)

That table 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.

²This is an *extremely* unrealistic assumption about names; your own name might not follow that rule, and even if yours does, 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.

A brief digression: Hello, server

You have by now learned a few things in CMSC 161 that we'll make use of in 160. If you haven't yet memorised how to do these things, *that's fine*, but you should be able to either look up how, or ask me.

- Make a directory `160` for all the work you'll do in this course
- Inside that directory, make a directory `lab2`
- Inside *that* directory, use `vim` to edit a file `hello.cpp`. Type in the full hello-world program that we learned last week; make at least one intentional typo that will cause a compiler error.
- Remember to save (write) the file when you're done!

Now, a new command:

```
compile hello.cpp
```

Because you intentionally made a typo, you should now see a compiler error... but only the first one. (I've set up the `compile` command to be nice that way.) Fix the error and recompile. Once it compiles without error, run this command:

```
./a.out
```

Then, you can hand in your work

```
handin cmsc160 lab2 .
```

Notice the single dot at the end, separated by a space from what came before it; that tells the `handin` command to hand in the entire current directory. (You could instead specify individual files or subdirectories, but in this case you really do want to hand in all the files in the current directory.)

If all goes well, you should get a message to that effect, and it will list all the files in the handin (probably just the `cpp` file, and `a.out`). If something is amiss, there will be an error message to tell you what to fix.

Typing in the test cases

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 work from the first part of this lab (on codeboard), 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. 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.³ At this time, the test is guaranteed to report that your

³There’s a lot to unpack there, and you don’t need to worry about it much (you can

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.

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

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`.

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.

AI policy and frequent submission

(so far no change from the Lab 1 version of this policy)

Some use of generative AI is fine, but you a) should not paste this assignment or type it verbatim into the AI prompt, and b) should not be asking the AI for the whole program all at once. (Just like you can ask for help from a human, but should not have them write the whole program for you!) If you get help from an AI chat OR from a person, you should note that in a program comment near whatever you got from them.

Relatedly, I expect that you'll click the Submit button relatively often, and I *require* that you do so at least 2–3 times over the course of working on the lab. (Note that codeboard only lets you submit if the program is successfully compiling, so that's another reason to want to try to compile and run after every few lines of code!) That gives me a version history and progress report on your work. Submissions that jump straight to a final, (near-)correct version with no intervening submissions along the way *will receive little or no credit*. ← scoring note!

This is a new policy that I'm experimenting with; let me know if you have any feedback.

Handing in

It's due as usual on Monday at 4pm. The codeboard parts are submitted with the green Submit button. On the server, hand in using the `handin` command, this time with assignment name `lab2`. Assuming you are in the directory where all your Lab 2 files are, you would type

```
handin cmsc160 lab2 .
```

(Don't forget the dot at the end; it says to hand in "this whole directory".)

Rubric

RUBRIC (Tentative)

- 1 Attendance at lab with preview done or question written down
- Covid levels**
- $\frac{1}{2}$ Documentation at top of file
- 1 Program runs, prompts, reads values
- 1 At least one valid `if` comparing input and printing level
- 1 Suitable `else if` structure
- 1 Comparison logic is correct to print correct levels in preview
- $\frac{1}{2}$ Nested `if` or equivalent in continuation
- $\frac{1}{2}$ Comparison logic is correct to print correct levels in continuation
- Checks order**
- $\frac{1}{2}$ Documentation
- 1 At least one pair of `.in/.expect` files w/ suitable contents
- $\frac{1}{2}$ Test case coverage, and TCs pass or are noted in readme*
- 1 Program compiles, runs, id's one case and responds correctly
- $\frac{1}{2}$ 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 Wednesday.

This document was written and prepared without the use of generative AI.