Lab 11
Sorting

17 November 2015

This week we'll continue working on the idea of putting values in order—sorting—and what has to be done to accomplish this. The "drill" is getting some bookkeeping out of the way so that we can dive right into the sorting algorithms during lab.

Drill: *Verifying* sortedness

1. Start files `sorting.h` and `sorting.cpp` with a function `is_in_order` that takes a `vector<string>` parameter and is intended (eventually) to determine whether the given vector is in sorted order or not. For now, though, just have it always return `false`. Make sure this much compiles by running

   ```
   compile -c sorting.cpp
   ```

   which, again, runs the compiler without running the linker (since we haven't defined a `main` function yet). (As *always*, try to compile as often as is reasonable, and start running your code as soon as it's feasible to do so.)

2. Write another program file `run_sort.cpp` that has a `main` function that reads lines from `cin` as long as there are any to be read, and puts them in a vector; and once it's done reading them, it calls `is_in_order` on that vector and prints the result. (Remember that that function is so far only returning false, and false prints as zero, so it's ok and expected that your program will just print zero regardless of its input!)

3. Start a unit test file `test_sorting.u` that will test whether `is_in_order` works correctly. Use the `test_stat_functions.u` file from Lab 6 as a model (on page 4 of the lab handout)—in this case the vector should be `vector<string>` rather than `vector<double>`, and should contain examples of `string` rather than examples of `double`, and you should have examples of vectors that *are* in order and vectors that *aren't* in order, to effectively test the function. In the actual test cases, you'll have lines that look something like this:

```
check (is_in_order(vec4)) expect true;
```

assuming you have a vector named `vec4` (you should probably pick a better name) and it is already in order. If it weren't in order, you would `expect false` instead.

If you have test cases that are meaningfully different, you can put them in separate `test` blocks if you like. Just make sure they're all part of the same `test suite` block (there should be only one of those for the whole file).

4. Return to `sorting.cpp` and edit `is_in_order` to do a little bit of work. If the given vector is of size 1 or shorter, immediately return true. If it's longer, compare (using the regular `<=` operator) the values of the first two elements of the given vector, and if the second is smaller than the first, return false; otherwise return true. Compile and run your test cases. Do they all succeed at this point? Should they? If all your tests succeed, continue to step 5. If any fail, continue to step 6.

5. If all your tests succeed in spite of the fact that you know your implementation of `is_in_order` is incomplete (it only compares two items), then that means your test cases aren't comprehensive enough. Add a test case that will catch this incompletely-implemented function; and compile it and run the test and see the test fail, indicating that `is_in_order` isn't done yet. Then continue to step 6.

6. At least one test is failing, so you have more work to do with `is_in_order`. Specifically, loop through indices and compare *every* adjacent pair in the vector to see if they're out of order. If you find an out-of-order pair, return false. If you get to the end of loop without finding an out-of-order pair, then you can return true.

7. If at any point you run the tests and they pass even though you're not done, go back to step 5: add a test case to verify whatever you forgot to check.

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 processes). 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 FOTD: Processes

In a terminal window, type

```
ps
```

and hit enter. This gives you a list of all processes running "inside" this terminal at this moment. One is `bash`, the command line shell itself; and the `ps` command itself is running, or rather, was running at the time the list was put together.

If you open another terminal window and run `ps` again, the output should be similar, but with different numbers in the first column ("PID") and second column ("TTY"). Run `ps` a third time, in either window, and you should notice that compared to the previous run, the PID for bash is the same but the PID for ps is different. That's because the same command shell is still running, but it was a new `ps` process each time; each new process gets its own unique process ID. Within a given window, the TTY values will remain the same—TTY is short for "teletype" or "teletypewriter", the original terminal, and indicates which terminal window is running the process.

Now type

```
ps -u username
```

except with your own login name there. This shows you all processes on this machine being run under your username. There are a bunch, because (if you've been following instructions) you have at least two windows open, and among the list you should see the `bash` processes you saw before (still associated with their same tty) and the current `ps` process.

Find the pid of the `bash` process in the other window. Type

```
kill XXXX
```

except replacing `XXXX` with the process's pid. This sends a "hang up" signal to that process, which (if it is well-behaved) will then quit.[1] You can use `ps` and `kill` together to cancel runaway processes, among other things. If a process is recalcitrant and ignores the interrupt, you can try

[1] Historically, this signal was invented to inform running processes that their user's connection had died, which in most cases meant the process should clean up after itself and then terminate.

```
kill -9 XXXX
```

which sends the process an unignorable "kill" signal. It's always better to try the hangup signal first, as it gives the process a chance to close any open files or save backups as appropriate.

While a process is running (in the foreground, as will be described below), hitting Ctrl-C is usually equivalent to sending the process a hangup signal. This doesn't apply to big text applications like Vim (or nearly any GUI app, once you start using those), but does apply to any C++ program you write.

Job control

Most command-line processes run by default in the "foreground". That means that when you execute the command, the shell waits for it to finish before giving you a prompt as to what to do next. There are two ways for processes associated with a terminal to not be in the foreground: suspending them and backgrounding them.

Most of you have seen accidentally how to suspend a process. In most situations, if you hit Ctrl-Z while a process (such as vim) is running, it will suspend itself. Suspending it makes it pause and returns you to the command line, but the process is still in memory, and can be reactivated. Try it: run `vim foo` (or whatever) and then hit Ctrl-Z. On the command line type

```
jobs
```

and you'll get a listing of non-foreground processes. You can put it back in the foreground by typing

```
fg
```

at this point. Do so, but then suspend it again.

The other sort of non-foreground process is a background process. That means that it's actually *still running*, but you can still use the command line in the meantime. One way to do this is when you execute a command. If you type

```
sleep 5
```

the command line will be unavailable for the five seconds this process is running. But if you type

```
sleep 5 &
```

it tells you the job number and pid assigned to the process and then lets you work on the command line. When it finishes, it lets you know it's done. Run `sleep 5 &` again, but this time run `jobs` right afterward; you should still have the suspended `vim` process, plus the running (but backgrounded) `sleep` process. In a moment, it finishes.

If you run `sleep 5` but then immediately hit Ctrl-Z, it suspends the process rather than backgrounding it. `jobs` will tell you two processes are now suspended. You can restart the process, but in the background, by typing

```
bg %2
```

(assuming the job number for the `sleep` process is `[2]`). Try this again (run `sleep 5`, suspend it, type `jobs`) and you can instead cancel the process entirely with

```
kill %2
```

Although `kill` requires an argument (either a pid or a job number with a percent sign), if you run `fg` or `bg` without an argument they will default to operating on `%1`, which is why `fg` by itself worked earlier.

Sorting things

We've been talking about various orders for `Point` values this week in lecture, and the code you'll write later in the lab will involve sorting `string` values, but for now I want to step back and have you work on putting things in order with actual physical objects—playing cards. For our purposes today we'll ignore the suit of the card and exclude aces (so we don't have to worry if they're high or low): sorting a list of cards will involve putting it in increasing order, from 2 to King.

The back page of this handout is a template that represents a vector of four playing cards, and I'll be giving you actual playing cards to put into that

"vector", and move around, and put in order. Pull that sheet off the packet so you can use it. Whenever I say to test a sorting algorithm, I want you to test at least three starting configurations:

- the cards are already in increasing order,
- the cards start out in *decreasing* order (and the sorting process should reverse them), and
- the cards start out in this order: second-highest, second-lowest, highest, lowest. (This order is a good "out of order" order that may help you understand the sorting process better.)

Also tear off the four index markers (labelled "i", "pass", etc) on the bottom of the rubric page. Use these to keep track of the current values of those variables when tracing the algorithms. (Not every index marker is used in every algorithm.)

A first attempt

With that in mind, try running the algorithm represented by the following pseudocode—which is meant as a first attempt to sort the cards but is not correct. In this listing, `cards` refers to the four-element vector represented by the sheet in front of you.

```
i ← 1
while i < size of cards
    if card at index i is lower than card at index i−1
        swap card at index i with card at index i−1
    i ← i + 2
```

Make sure to use the "i" marker to keep track of the value of i as you trace through the algorithm on the three test cases. In the space below, make note of which parts of the algorithm seem to work, and what's missing:

A revision

Noting that the previous algorithm doesn't fully sort the cards but does at least make them "more sorted" than before, someone suggests refining the algorithm in this way:

```
pass ← 0
while pass < size of cards
    i ← 1
    while i < size of cards
        if card at index i is lower than card at index i−1
            swap card at index i with card at index i−1
        i ← i + 2
    pass ← pass + 1
```

Run this version of the algorithm (this time keeping track of the values of both i and pass) on the test cases.

Does it help? Does it fix the problems you noted on the previous page? It still doesn't quite work, but it may suggest a strategy that does; devise a minor modification you could make to this pseudocode to make it work, write down the change you made, and test the revised algorithm too to confirm that it works.

A different approach

Meanwhile, a different person is proposing the following, rather different, algorithm:

```
target ← size of cards − 1
while target ≥ 0
    i ← 0
    highestindex ← 0
    while i < size of cards
        if card at index i is higher than card at highestindex
            highestindex ← i
        i ← i + 1
    swap card at index highestindex with card at index target
    target ← target − 1
```

As before: run this version of the algorithm (tracking target, highestindex, and i) on the three test cases; and make note of what parts of the strategy are successful and what isn't working:

Devise a modification to this algorithm to make it work, write down the change you made, and test the revised algorithm to confirm that it works.

Note for handing in: I want to see a (short) note in your readme that indicates, for the playing card algorithms in this section of the lab, what the problem was and how you fixed it. This can be just a sentence or two per algorithm.

Translate it into code

The sorting strategy used in the "first attempt" and "revision" above is generally called a *bubble sort*, and the "different approach" is known as a *selection sort*. What you'll do now is take your fixed versions of each algorithm, and translate them into functions called `bubble_sort` and `selection_sort` that you write in the files you started during the drill. Each of them should take a `vector<string>` and compute and return a vector that has the same values, but in sorted order.

In the test file, the easiest way to write tests for these is to take the return value of the sorting function and then ask questions about the results, starting with the function you wrote in the drill:

```
vector<string> result3 = bubble_sort(vec3);
check (is_in_order(result3)) expect true;
```

Assuming that `vec3` is defined earlier in the `test` block (whether it's in order or not!—but seriously, you should pick a better name for your example than `vec3`), this test code says that the result that `bubble_sort` returns should itself be in order.

You should think about why it isn't completely sufficient to use `is_in_order` as the only way you test the results of your sort functions—and what other questions you should ask to more effectively test them.

Note that your tests for `selection_sort` can be almost identical to your tests for `bubble_sort`, except for calling the other function instead. Also, don't forget that you can get the points for good testing even if the thing they're testing isn't working yet (and thus the test "fails"—which is good, because it reminds you of what work is left to do).

For now, don't forget to return the sorted result from each function. Later this week, we'll see another way that we could set up these functions, and when we do, you *may*, but *don't have to*, use that instead.

There is a builtin function `swap` that does just what you'd want it to when you give it two vector elements that you'd want to swap:

```
swap (cards[1], cards[3]);
```

would swap the value at index 1 with the value at index 3, should you be so inclined.

The program in `run_sort.cpp` should read input and print whether it's sorted (as it currently does), then call one of the sort functions you wrote (either one—if only one works, use that one!), and print the elements in the resulting vector, and the result of another call to `is_in_order`.

Handing in

As usual, use the `handin` program. Designate this as `lab11`. Hand it in by 4pm on Monday, 23 November.

Rubric (tentative)

RUBRIC

General

1 Attendance at lab with drill done or question written down
½ Readme with clear instructions on compiling, running, testing
1 ...and info on errors in/how to fix playing card sorts

Drill

1 All four files set up correctly, correct header for `is_in_order`
½ Test cases for `is_in_order` include ≥ 1 that fails alg in drill step 4
1 `is_in_order` correct
½ `run_sort` runs at least as described in drill step 2

Implementation

1 Both sort functions are well-tested in `.u` file
1 `bubble_sort` implemented to match version on p7
½ ...and fixed to work correctly
1 `selection_sort` implemented to match version on p7
½ ...and fixed to work correctly
½ `run_sort` runs as described on p9