EECS 281 Project 0 -- UNGRADED

"Due" Tuesday 9/1 by 11:59pm

The goal of this project is to help you become more familiar with command line processing, as well as how a `vector` works and the `vector` member functions `.resize()` and `.reserve()`. Reading the Lab01-Prelab document will help with Project 0.

Command Line

The `getopt_long()` function has several parameters, but the two most important are an array of structures and a double-quoted string. In the Lab01-Prelab document, the array is declared like this:

```
static struct option longopts[] = {
    { "add",    no_argument,       nullptr, 'a' },
    { "delete", required_argument, nullptr, 'd' },
    { nullptr,  0,                 nullptr, '\0' }
};
```

This says that there are two valid command line options, `"add"` and `"delete"`, and that they have short forms `'a'` and `'d'`, respectively. The third line is like a `'\0'` for the array of options: it allows `getopt_long()` to know when it's reached the end. The second field on each line indicates whether that option is followed by something else that is not an option. The `no_argument` means that option is not followed by anything else, where as `required_argument` means it is. For example, these would be valid command lines for the prelab sample program:

```
./lab1-sample --delete something
./lab1-sample -a
./lab1-sample --add -d test
```

Notice that delete/d is always followed by another "thing" (thus that thing is required), whereas add/a is never followed by another "thing" (except possibly the start of a new option).

The second part is the double-quoted string. It's redundant to have both the array of structures and the double-quoted string, but we're stuck with it. The sample program string is `"ad:"`. The single-letter options from the array are repeated, and any `required_argument` has a colon following it.

Vector Size

Think of a `vector` as having two sizes: how many items are stored in it right now (referred to as the size), and the capacity before it has to resize (this is the size of the dynamic memory that it allocated via `new`). If you create a `vector` and just use `.push_back()`, items are added, the size increases linearly, but the capacity doubles any time there is not enough room (some implementations might multiply the size by a different factor, but it's always multiplicative). So if you keep using `.push_back()`, an empty `vector` would grow from 0/0 (size/capacity) to 1/1, 2/2, 3/4 (notice that one space is wasted), 4/4, and then 5/8 (the capacity doubles from 4 to 8, then there's room to add the latest item). Whenever the `vector` grows by doubling, here's what has to happen:

1) A new block of dynamic memory is allocated, of the new capacity
2) The existing elements are copied from the old block to the new block
3) The old block of dynamic memory is deleted
4) The internal pointer is changed to point to the new block of dynamic memory
5) The`.push_back()` can finish

Step (2) takes time, and you can end up with wasted memory. Consider 1025 elements: the size doubles 1, 2, 4, …, 512, 1024, 2048. But now only 1025 elements are used out of 2048 allocated, which is almost 50% wasted memory. If you know ahead of time how much data you will need room for, you can use the `.resize()` or `.reserve()` member functions to change the size of the `vector` before reading. You will know that the size is exactly right, with no wasted memory.

The `.resize()` member function changes both the current size and the capacity. Once this is done, you can use `[]` to access any valid index in the range `[0, size)`. If you use .resize() and then immediately use .push_back(), you will end up with extra elements. For example, if I `.resize()` a `vector` of integers to 10, it has 10 copies of the value 0. If I then `.push_back()` the value 25, I have 10 copies of 0 followed by the value 25.

The `.reserve()` member function changes only the capacity, leaving the current size unchanged. Thus to add more elements you would need to use `.push_back()`. If you use `.reserve(10)` and then immediately try to access the data at index 0 (using square brackets), it's an invalid access.

Finishing Project 0

You should first complete the `getMode()` function; until that is done none of the other functions can be called. After that's done, pick one of the three functions that read data and complete it. You can use the two input files provided to test that your program is working. The file

`sample-n.txt` is intended to be used with `--mode nosize`, while the file `sample-r.txt` will work with the `resize` or `reserve` modes. When your program is working, the correct answers are an average of 13.39, and median of 12.70 (both files and any valid command line flag should give the same results).

Autograder (9 points total over 9 test cases)

After you have your program working, typing the command "`make fullsubmit`" (again without quotes) will produce a tarball name `fullsubmit.tar.gz`, which you can then transfer back to your PC and upload to the autograder for further testing and scoring. There are 9 tests on the autograder, each test has a two letter name: S|M|L (**S**mall, **M**edium, or **L**arge input file), followed by N|V|Z (**N**osize, reser**V**e, or resi**Z**e).

Late Days

We encourage you to use your late days on Project 0, just to see how the system works. Since everyone has 2 late days, you could extend the project by 1 or even 2 days. On the third day after the due date, we'll modify the database to restore everyone's late days, before any actual assignments are due.