# EECS 281 – Fall 2022

# Programming Project 0

# *Statistics* [UNGRADED]

"Due" Thursday, September 8, 2022 at 11:59 PM

## Overview

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.

Not only will this help you with command-line processing that is much more complicated than you did in EECS 280, it will also familiarize you with the EECS 281 Autograder.

There are 5 videos to help you get started on our EECS 281 YouTube Channel.

Pick one of the following:

- Using a Mac and Xcode
- Using Windows, Visual Studio and the WSL
- Using Windows and VSCode w/ WSL

Pick two of the following:

- Project 0 tutorial
- Using our Makefile

If you're using Visual Studio, make sure it is version 2019 or 2022 and fully updated, otherwise you may have problems with `getline()` and input files. This issue exists because Windows uses two characters to indicate end-of-line, where all other operating systems use one character. This has been a challenge when working in Microsoft environments for decades, though current versions of Visual Studio handle this issue more seamlessly.

## Starter Files

The tarball, Project0-starter.tar.gz, includes starter code in project0.cpp, a Makefile, and test input files sample-n.txt, and sample-r.txt.

## 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 the null terminator for strings (`'\0'`) but 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, while `required_argument` means it must be. 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" that is not a short or long option (this is a required option), whereas `--add/-a` can only be followed by another option or the end of the command.

The second part is the double-quoted string. It may seem redundant to have both the array of structures and the double-quoted string, but `getopt_long()` uses `getopt()` so both are needed. The sample program string is `"ad:"`. The single-letter options from the array are used here, and any `required_argument` has a colon (:) following it. Any single letter option that isn't followed by a colon is considered `no_argument`. Outside the scope of the course, a single letter option followed by a double colon (::) is considered to have an `optional_argument`. For more information, read the manpage (https://linux.die.net/man/3/getopt_long).

## Vector Size and Capacity

Think of a `vector<>` as having two measurements: how many items are stored in it currently (referred to as the size), and the total number of items it can hold before no more items can be added without resizing (referred to as the capacity). These measurements may be equal if the vector is "full", but the size can never be greater than the capacity. 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.

Complete documentation of the STL `vector<>` can be found at https://cplusplus.com/reference/vector/vector/.

# Finishing Project 0

## Coding Plan

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 `--mode resize` or `--mode reserve`. When your program is working, the correct answers are an average of 13.39, and a median of 12.70 (both files and any valid command line flag should give the same results).

## Project Identifier

***You MUST include the project identifier at the top of every file you submit to the autograder as a comment. This includes all source files, header files, and your Makefile (search for the TODO in the Makefile, and remember that it uses # for comments not //):***

Add to all source code and header files

```
// Project Identifier: 39E995D00DDE0519FDF5506EED902869AEC1C39E
```

Add to Makefile

```
# Project Identifier: 39E995D00DDE0519FDF5506EED902869AEC1C39E
```

## Submitting to the Autograder

Before submitting to the autograder, be sure that:

- The executable should be named `project0` (look in the `TODO` section of the `Makefile`)
- Every source code and header file contains the following project identifier in a comment at the top of the file: `39E995D00DDE0519FDF5506EED902869AEC1C39E`
- The Makefile must also have this identifier (already done for you for Project 0).
- DO NOT copy the above identifier from the PDF! It might contain hidden characters. For Project 0, the identifier is already included in the provided `project0.cpp` file. Future projects will have it in a text file for you to copy to your code.

After you have your program working, typing

```
$ make fullsubmit
```

will produce a tarball name `fullsubmit.tar.gz`, which you can then upload to the autograder for further testing and scoring. There are 9 tests (9 points total) 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 test using your late days on Project 0, just to see how the system works. Since everyone has at least 2 late days (graduate students in EECS 403 have 3), you could extend the project by 1 or even 2 days. A few days after the due date, we'll modify the database to restore everyone's late days, before any actual assignments are due.