Tutorial 5
Managing Memory

Learning Objectives

After this tutorial, you will be able to:

- Use dynamic memory when writing C programs. Specifically:
  - Allocate Memory
  - Pass Parameters by Reference
  - Free Memory No Longer Needed
- Use valgrind to debug memory leaks

Tutorial

In order to receive full marks, you must complete everything in the "Tutorial" section. Part marks may be provided if good progress was made toward completing the tutorial, at TA discretion.

1. Download the file `T05.tgz` from the tutorial module on Brightspace. Untar and read through the files.

   Rather than tar a file (making a `.tar` file) and *then* `gzip` it (making a `.tar.gz` file), you can create one file (a `.tgz` file) that does both! You can extract the files in a `.tgz` file with an appropriate option using `tar`.

2. Compile the program with the `-g` flag to include debugging information; this is helpful when using `valgrind`.

   ```
   gcc -Wall -std=c99 -o t05 t05.c t05util.c -g
   ```

   Note that in this tutorial, we are compiling **two** source files. What happens if we only compile `t05.c`?
   We will be covering more ways of compiling multiple files shortly, but you can just follow along for now!

3. Run `valgrind` on the executable output `t05` to check for memory leaks:

   ```
   valgrind ./t05
   ```

4. Fix `t05.c` to remove the memory leaks.

   When you run valgrind without any options, you will notice that it gives a nice summary of the state of the memory at exit, but if there are leaks, it will recommend that you rerun `valgrind` with the `--leak-check=full` option. Try that out, and carefully read the output to see if it helps you to hunt down your memory leaks.

   Note: calloc(1, *size*) is like malloc(*size*) in that they both reserve a single "size" of space, but calloc() will make sure the memory that is allocated is all zeroes so that we don't accidentally read garbage data. It is beneficial if we can't guarantee that all of our memory will be properly initialized, as we can then check whether it's initialized or not.

5. Implement the functions `growArray()` and `addStudent()` as prototyped. If `addStudent()` is called and the array is full, use `growArray()` to increase the array's capacity before adding the student.
   Hint: The "array" in StudentArray is a pointer to a dynamically allocated block of memory! Is there any reason we couldn't point it to a large, newly allocated block?

6. Use `valgrind` to ensure that the changed program has no memory leaks.

Exercises

These exercises are optional, but are provided to give you some additional practice working with common operations that you will encounter while using pointers that might be tricky when you first work with them.

1. Write a function `removeStudent(StudentArray* stuArray, int index)` which removes the student at the given index.

   What impact does the removal have on `stuArray->count`? Make the necessary changes to `stuArray` so that `printArray()` still functions correctly after removal.

2. Alter the function `addStudent()` so that `stuArray` will always be sorted alphabetically by name.

Saving and Submission

In-Person Attendance: Submission is optional, as you can be checked off in class, but you must still complete the required portion of the tutorial for marks.
Asynchronous: If you are completing the work on your own at home, make sure to follow the submission requirements.

1. Create an archive to submit your files, and include all of the files that you worked with in the tutorial.
   a. They may be submitted as a .tar, .tar.gz, or .tgz file
   b. Make sure to include all of the code needed to run your program

2. For **full marks** you should have completed all problems in the Tutorial section (and they should be seen upon running the program) without any memory leaks on exit. The exercises are optional, but highly recommended for the extra challenge and exploration of these concepts.

3. For **part-marks** you should have attempted to complete most of the tutorial. Grades for part-marks will be at TA discretion.