Your programming tasks throughout the course are demanding. In going through these exercises, you will become a very strong developer. Strong developer skills and strong debugging skills are one and the same! This document aims to serve as a manual for some helpful tips and tricks in two of the most common command-line debugging tools: `gdb` and `lldb`. These are the best suited for debugging C/C++ based repositories, so they are the recommended tools for this course, but you are welcome to use other debugging methods as you see fit.

1 Sample Usage

Suppose we have the following `buggy.cc` program:

```
#include <chrono>
#include <cstdlib>
#include <iostream>

#define ARR_SIZE 1024
#define NUM_OPS 16384

using namespace std;

int
main(int argc, char **argv)
{
    int arr[ARR_SIZE];
    memset(arr, 0, ARR_SIZE);

    auto t1 = chrono::system_clock::now();

    for (int i = 0; i < NUM_OPS; i++) {
        int index = rand();
        arr[index]++;
    }

    auto t2 = chrono::system_clock::now();
    auto execution_time = chrono::duration_cast<chrono::milliseconds>(t2 - t1);

    cout << "execution time: " << execution_time.count() << endl;

    return 0;
}
```

We can compile this program by calling `g++ -std=c++14 -g -O0 -o buggy buggy.cc`, and if we run the program (`./buggy`) we will get an exception (`Segmentation fault`).

All hope is not lost! We can figure out the cause of the fault by running the program in a debugger and examining the state.

2 Compiling a Debuggable Program

A few things to note about our compile line `g++ -std=c++14 -g -O0 -o buggy buggy.cc`

- Notice our compiler is `g++` – this enables us to compile C++ programs (either the `.cpp` or `.cc` extension).
- Just like when writing C programs compiled `gcc`, the option `-o <name>` creates a binary with the name.

	`gdb`	`lldb`
launch the debugger	`gdb --args <cmd> <args>`	`lldb -- <cmd> <args>`
run a program	`r`	`r`
execute next line in function	`n`	`n`
step into next line	`s`	`s`
set a breakpoint at a line	`b <file>:<line>`	`b <file>:<line>`
continue execution after stop	`c`	`c`
ignore a breakpoint n times	`ignore <bp num> <n>`	`breakpoint modify -i <n> <bp num>`
command when breakpoint is hit	`command <bp num>`	`breakpoint command add <bp num>`
add condition to breakpoint	`condition <bp num>`	`breakpoint modify -c <bp num>`
show current breakpoints	`info b`	`breakpoint list`
print variable	`p <var>`	`p <var>`
show call trace	`bt`	`bt`
move up/down a function call	`up` / `down`	`up` / `down`
go to frame n	`f <n>`	`f <n>`

Table 1: (not-at-all-comprehensive) Debugger Command Cheat Sheet

- The option `-std=c++14` tells the compiler which version of C++ we are using. In this class, the typical options will be `-std=c++14` or `-stc++17`.
- Just like when writing C programs compiled with `gcc`, the option `-g` adds debug symbols to the binary that allow debugging tools like `gdb` and `lldb` to display information about the program context. Without this, `gdb` and `lldb` may not be able to present meaningful information when stopping at breakpoints, etc. What does this mean about the speed of using `-g` versus omitting it?
- Just like when writing C programs compiled with `gcc`, the option `-O<0, 1, 2, 3>` specifies the degree of compiler optimizations that should be used to produce the binary (where `-O0` is the least optimized and `-O3` is the most optimized). When debugging, we use `-O0` to ensure the program is not reordered, rewritten, etc.

In general, add the `-g` and `-O0` flags to your compilation line, and you will be good to go to run your program in a debugger! For more, see `man gcc`.

3 Starting a Program

Depending on your preference, you may use `gdb` or `lldb` to debug your programs. The two tools have largely similar functionality, but use different syntax. To start the program in the debugger, you will need to launch the debugger with your program and the input arguments for the program. Then, run it! The program will either run to the end of its execution (in which case the debugger will print a message saying that the program exited cleanly; exit code 0) or until a signal is raised (e.g., an error). In the case of our `buggy` executable, the debugger will stop the program on line 20 if the index is out of bounds. From here, we can examine the state around the line that crashed to deduce what is going wrong.

This document will be updated throughout the term. Happy debugging!