ECE/CS 250: A Guide to Debugging C

Common issues:

- For programs that require command line arguments, be sure to actually provide them while manually running the program. If you don't, you'll encounter a segmentation fault because your program tried to access elements in **`argv`** beyond **`argv`**'s bounds.
- For programs where one or more command line arguments are names of (or paths to) files, make sure those files actually exist in the location specified by the provided command line argument(s).
- For recursive programs, make sure you've implemented base case checks, and that you're actually modifying the input arguments in a way that the base cases will eventually be reached. A common mistake is to overshoot beyond the base case.
- Don't define large arrays (on the order of MBs) on the stack. Do so on the heap or global data
- Make sure you free pointers that you allocate. Make sure you don't free pointers that have already been freed. Make sure you actually allocate memory to pointers before using them.
- **If you have taken care of all these common mistakes and bugs still persist, you need to debug your program (in runtime) using the GNU Debugger (aka GDB).**

Debugging with GDB:

GDB allows you to run your programs within GDB's environment, and debug your programs by pausing them at breakpoints, stepping through line-by-line etc., all in real time. It also allows you to view variables' real time contents, the contents of memory in real time, the values in registers (you'll learn about these when talking about assembly) and most importantly for our purposes, *the exact location of the occurrence of a segmentation fault.* Follow the steps below to use GDB.

- In order to make a program capable of being debugged by GDB, make sure you compile it with the **`-g`** flag e.g. **`gcc -g -o outputfilename sourcefilename.c`**
- To run a program using GDB, launch it as such: **`gdb /path/to/your/program`** This will load **`program.o`** in GDB's environment.
- To run the program you've just loaded, use the **`run`** command in GDB. Many commands in GDB come with shorthands. Instead of typing **`run`** every time, you may use the shorthand **`r`.** To run a program with command line arguments, run it as such: **`r arg1 arg2`** … **`argn`**
- To find the origin of a segmentation fault, follow the following steps:
  - When you use the run command above, at some point your program should crash.
  - At this point, type the command **`backtrace`** (shorthand **`bt`**) and hit return. GDB will show you exactly what line caused the segmentation fault.
  - At this point you should be able to reason about why that particular line caused a segmentation fault and remedy the issue.
- To step through your code line-by-line, follow the following steps:

    - First, you need to set a breakpoint. A breakpoint is a point in your code up to which the program will execute in one go. At that point, you will gain control and you can step through it line by line.
    - To set a breakpoint, use the **`break`** command (shorthand **`b`**) as such: **`b main`**. Instead of **`main`**, the argument can be any function name (e.g. "main," "foo," "bar" etc.) or even a line number.
    - Once you've hit a breakpoint, execution will pause, and you'll have the ability to manually control execution using the following commands:
        - Use the **`step`** command to go to the next line in your source code.
        - The **`next`** command is almost the same as **`step`**, but unlike **`step`**, **`next`** will treat entire function calls as a single instruction instead of jumping into functions.
        - The **`continue`** command will return control to the computer and finish the execution of the entire program ass usual.
- To view variables' contents in real time, use the **`print`** command as such: **`print var`** where **`var`** is some variable. It can also be a memory location such as **`0x12345678`**.
- The **`display`** command (Usage: **`display var`**) does the same thing as **`print`**, except that **`display`** will continue to automatically print the values of the requested variable/memory location every time you step through. To turn this back off, use the **`undisplay`** command (Usage: **`undisplay var`**).

Debugging with Valgrind:

Valgrind allows you to discover and debug issues related to memory consumption, memory leakage and other pointer misuse.

- **Memory usage:**
    - To check your programs for basic memory usage stats, simply run your program as such: **`valgrind ./prog`**
    - To pass argument(s) run Valgrind like so: **`valgrind ./prog arg1 arg2 … argN`**
    - When the output is printed out, look for a section named "HEAP SUMMARY." In this section, you will see how many allocs and frees were called in your program, and ideally these numbers should be equal. If not, **you have a memory leak.**
- **Memory leaks:**
    - If you've done the things above, and you've determined that your program has memory leaks in it, you can run Valgrind with the following parameters:

      ```
      valgrind --tool=memcheck --leak-check=yes ./prog
      ```
    - This will show you all the allocation functions that didn't have subsequent frees.
- **Other bugs:**
    - If you have bugs along the lines of "Conditional jump or move depends on uninitialized value(s)," then you must make sure all your pointers are initialized to some values. This

includes pointers (such as your linked list nodes' next pointers) whose values are checked for being NULL.

- If you're unable to pinpoint the exact source of pointers that cause such issues, you may use Valgrind with the track-origins flag as such: **`valgrind --track-origins=yes ./prog`**