Debugging with GDB and Valgrind

Plenty of excellent resources focused on GDB can be found at the CSE351 GDB page :
https://courses.cs.washington.edu/courses/cse351/18wi/gdb/

To use GDB and Valgrind, compile your C program using the "-g" flag

Starting GDB

```
bash$ gdb -tui <program file name>
```

GDB Commands

*Shortcuts for commands are **bolded**.*
Shortcuts with more than 1 character may or may not include spaces in between chars.

[IN GDB] Controlling Program Execution

- **r**un <args> — Run your program with args
- **n**ext — Go to next instruction (source line) but don't dive into functions
- **s**tep — Go to next instruction (source line), diving into function
- **f**inish — Continue until the current function returns

[IN GDB] Setting Breakpoints and Continuing

- **b**reak <where> — Set a new breakpoint
- **i**nfo **b**reakpoints — Print informations about the break and watchpoints
- **c**ontinue — Continue normal execution

[IN GDB] Understanding the Stack and the Current Function

- list — Shows the current or given source context
- **i**nfo **a**rgs — Print the arguments to the function of the current stack frame
- **i**nfo **l**ocals — Print the local variables in the currently selected stack frame
- **i**nfo **f**rame — Print information about the current stack frame
- frame <frame#> — Select the stack frame to operate on

[IN GDB] Displaying Memory, Variable Values, and the Call Stack

- **p**rint /format <what> — Print content of variable/memory location/register
- e**x**amine /format <what> — Treat <what> as a pointer; print the content at the address it points to
- **b**ack**t**race — Show call stack

[IN GDB] Keeping Track of Variables over Time

- **d**isplay /format — Like "print", but print the information after each stepping instruction
- watch <where> — Set a new watchpoint (break when a variable changes value)

[IN GDB] Add visuals that display your code and register values as your code executes

- layout reg — Displays all registers and their current values.
- layout src — Displays the C code your program is currently executing.
- layout asm — Displays the assembly code your program is currently executing.
- layout split — Displays the C and assembly code your program is currently executing.

Use *valgrind* to detect memory leaks

bash$ `valgrind --leak-check=full <command to start your program>`

- Throughout this course, we will be using `valgrind` to evaluate your code for memory leaks. `valgrind` examines your program and identifies any possible memory leaks during execution. Make sure to run your code through `valgrind` before submitting if your code uses `malloc`!

Finding a String In All of Your Source Code

bash$ `grep -r -n "target-string" file_or_directory`

- Search the "target-string" in the specified file or directory. Useful for finding provided function Implementation.

Common Memory Errors:

These are some of the more common errors related to memory management.

- Use of uninitialized memory
- Reading/writing memory after it has been freed – Dangling pointers
- Reading/writing to the end of malloc'd blocks
- Reading/writing to inappropriate areas on the stack
- Memory leaks where pointers to malloc'd blocks are lost
- Mismatched use of malloc/new/new[] vs free/delete/delete[]
- Forgetting to check for NULL or dereferencing NULL