A lot of the information here was taken from EECS 201 and EECS 482.
Text in `{curly brackets}` is optional. Text in **`<angled brackets>`** are templates, not literal.

Print debugging a C or C++ file

1. At the top of the file, write this code:

```
#ifdef DEBUG_PRINT
#define debugprintf(fmt, ...) printf("DEBUG: " fmt, ##__VA_ARGS__)
#else
#define debugprint(fmt, ...)
#endif
```

2. Wherever you want to debug print, write:

```
debugprintf(<some_string>, <some_vars>);
```

3. After you compile the file (e.g $ gcc file_name.c), if you want debug printing, run:

```
$ ./<executable_name> -DDEBUG_PRINT <file_name>.c
```

If you don't want debug printing, exclude **`-DDEBUG_PRINT`** when running.

Logging levels

If you want, you can specify the seriousness of the debug information you're printing by categorizing it as one of the following:

- Fatal- errors that leave the program unable to continue; you must stop now.
- Error- general errors that aren't fatal
- Warning- potential things that are suspect
- Info- general milestones
- Debug- where you can give some more details
- Verbose/trace- where you provide a lot of details

When you run your program you can set the max verbosity to only log information that is above the minimum seriousness threshold (for example, only print warnings or more serious).

gdb

gdb is an interactive command line debugger.

When compiling your C or C++ program, add the **`-g`** flag to add debugging symbols (ex: **`$ gcc -g main.c`**).

You can invoke gdb by running **`$ gdb <executable>`**.

Here are commands that can be run in gdb:

Command	Description
(gdb) q{uit}	Quit the interactive debugger
(gdb) k{ill}	Kill
enter key	Repeat the previous command
(gdb) b{reak} **<func_name>**	Set a breakpoint at **<func_name>**
(gdb) b{reak} **<file_name>:<line_num>**	Set a breakpoint at **<line_num>**. Note that you can make this and the previous breakpoint conditional, for example: **(gdb) b main.cpp:21 if myVar < 4**
(gdb) i{nfo} **b**{reak}	List all currently set breakpoints, including their number, whether they're enabled, and where they are.
(gdb) dis{able} **<breakpoint_num>**	Disable breakpoint number **<breakpoint_num>**
(gdb) en{able} **<breakpoint_num>**	Re-enable breakpoint number **<breakpoint_num>**
(gdb) del{ete} **<breakpoint_num>**	Delete breakpoint number **<breakpoint_num>**
(gdb) watch <var_name>	Watch **<var_name>** (that is, break whenever **<var_name>** changes). This also tells you how **<var_name>** changed.
(gdb) r{un} **<args>**	Run the executable with the provided **<args>**
(gdb) c{ontinue}	Continue execution of the program after pausing at a breakpoint
(gdb) n{ext}	Execute the current line of code, and move on to the next line
(gdb) s{tep}	Step into (enter) any functions that are in the current line
(gdb) fin{ish}	Run until you return from the currently stepped-into function
(gdb) p{rint} **<var_name>**	Print the current value of variable **<var_name>**
(gdb) l{ist}	List the lines of code around where you're currently at
(gdb) b{ack}**t**{race}	Backtrace; show information about stack frames, including their number.
(gdb) f{rame} {<frame_num>}	View current line. If **<frame_num>** is specified, switch over to and print information about stack frame **<frame_num>**. You can go back to where you were by running **(gdb) frame 0**.

(gdb) u{p} **<num_frames>**	Less recent frames
(gdb) d{own} **<num_frames>**	More recent frames
(gdb) handle SIGUSR1 nostop noprint	Use this for EECS 482 projects.
(gdb) i{nfo} **thr**{eads}	List threads
(gdb) thr{ead} **<thread_num>**	Switch to thread **<thread_num>**

Core dumps

Core dumps are files created when a program crashes, populated with the state of the program's memory at the instant the crash occurred. An easy way to ensure your program crashes is by using **`assert()`** statements. The core dump file's name usually starts with "core" (ex: core.12345).

Instead of running a program in GDB, you can tell GDB to load your core dump. When viewing a core dump, you can do everything you normally do in GDB that doesn't involve running the program (e.g. step, next, etc.).

Here's how to create a core dump:

```
$ ulimit -c unlimited
$ ./<executable_name>
Segmentation fault (core dumped)
$ gdb ./<executable_name> -c <core_dump>
```

Core dumps can be created in various locations. If they don't save to your current working directory by default, you can configure them to save there this session with:

```
$ echo "$(pwd)/core-%e-%p" | sudo tee /proc/sys/kernel/core_pattern
```

Valgrind

Valgrind is dynamic- it runs when you run your program- and can check for memory leaks use-after-frees, invalid read and writes, and use of uninitialized variables, among other things.

When compiling your C or C++ program, add the **`-g`** flag to add debugging symbols (ex: **`$ gcc -g main.c`**).

You can invoke Valgrind by running:

```
$ valgrind ./<executable_name> <args>
```

For more detailed information, you can run:

```
$ valgrind --leak-check=full ./<executable_name> <args>
```

I recommend using the --leak-check=full flag. You'll be given more information including how many allocations, how many frees, which line in the program a memory leak/bad read/bad write occurred, etc. Without the flag, you're basically just told whether there are or aren't any memory issues. A program is good if there are 0 memory leaks and 0 errors.

Note: you can see how much memory a process is using by making a new terminal and running **`$ top`** in it.