Exercise 20. Advanced Debugging Techniques

I've already taught you about my awesome debug macros, and you've been using them. When I debug code I use the `debug()` macro almost exclusively to analyze what's going on and track down the problem. In this exercise, I'm going to teach you the basics of using GDB to inspect a simple program that runs and doesn't exit. You'll learn how to use GDB to attach to a running process, stop it, and see what's happening. After that, I'll give you some little tips and tricks that you can use with GDB.

This is another video-focused exercise where I show you advanced debugging tricks with my technique. The discussion below reinforces the video, so watch the video first. Debugging will be much easier to learn by watching me do it first.

Debug Printing versus GDB

I approach debugging primarily with a "scientific method" style: I come up with possible causes and then rule them out or prove that they cause the defect. The problem many programmers have with this approach is that they feel like it will slow them down. They panic and rush to solve the bug, but in their rush they fail to notice that they're really just flailing around and gathering no useful information. I find that logging (debug printing) forces me to solve a bug scientifically, and it's also just easier to gather information in most situations.

In addition, I have these reasons for using debug printing as my primary debugging tool:

- You see an entire tracing of a program's execution with debug printing of variables, which lets you track how things are going wrong. With GDB, you have to place `watch` and `debug` statements all over the place for everything you want, and it's difficult to get a solid trace of the execution.
- The debug prints can stay in the code, and when you need them, you can recompile and they come back. With GDB, you have to configure the same information uniquely for every defect you have to hunt down.
- It's easier to turn on debug logging on a server that's not working right, and then inspect the logs while it runs to see what's going on. System administrators know how to handle logging, but they don't

know how to use GDB.

- Printing things is just easier. Debuggers are always obtuse and weird with their own quirky interfaces and inconsistencies. There's nothing complicated about `debug("Yo, dis right? %d", my_stuff);`.
- When you write debug prints to find a defect, you're forced to actually analyze the code and use the scientific method. You can think of debug usage as, "I hypothesize that the code is broken here." Then when you run it, you get your hypothesis tested, and if it's not broken, then you can move to another part where it could be. This may seem like it takes longer, but it's actually faster because you go through a process of differential diagnosis and rule out possible causes until you find the real one.
- Debug printing works better with unit testing. You can actually just compile the debugs while you work, and when a unit test explodes, just go look at the logs at any time. With GDB, you'd have to rerun the unit test under GDB and then trace through it to see what's going on.

Despite all of these reasons that I rely on `debug` over `GDB`, I still use `GDB` in a few situations, and I think you should have any tool that helps you get your work done. Sometimes, you just have to connect to a broken program and poke around. Or, maybe you've got a server that's crashing and you can only get at core files to see why. In these and a few other cases, GDB is the way to go, and it's always good to have as many tools as possible to help solve problems.

Here's a breakdown of when I use GDB versus Valgrind versus debug printing:

- I use Valgrind to catch all memory errors. I use GDB if Valgrind is having problems or if using Valgrind would slow the program down too much.
- I use print with debug to diagnose and fix defects related to logic or usage. This amounts to about 90% of the defects after you start using Valgrind.
- I use GDB for the remaining mysteriously weird stuff or emergency situations to gather information. If Valgrind isn't turning anything up, and I can't even print out the information that I need, then I bust out GDB and start poking around. My use of GDB in this case is entirely to gather information. Once I have an idea of what's going

on, I'll go back to writing a unit test to cause the defect, and then do print statements to find out why.

A Debugging Strategy

This process will actually work with any debugging technique you're using. I'm going to describe it in terms of using `GDB` since it seems people skip this process the most when using debuggers. Use this for every bug until you only need it on the very difficult ones.

- Start a little text file called `notes.txt` and use it as a kind of lab notes for ideas, bugs, problems, and so on.
- Before you use `GDB`, write out the bug you're going to fix and what could be causing it.
- For each cause, write out the files and functions where you think the cause is coming from, or just write that you don't know.
- Now start `GDB` and pick the first possible cause with good file and function variables and set breakpoints there.
- Use `GDB` to then run the program and confirm whether that is the cause. The best way is to see if you can use the `set` command to either fix the program easily or cause the error immediately.
- If this isn't the cause, then mark in the `notes.txt` that it wasn't, and why. Move on to the next possible cause that's easiest to debug, and keep adding information.

In case you haven't noticed, this is basically the scientific method. You write down a set of hypotheses, then you use debugging to prove or disprove them. This gives you insight into more possible causes and eventually you find it. This process helps you avoid going over the same possible causes repeatedly after you've found that they aren't possible.

You can also do this with debug printing, the only difference is that you actually write out your hypotheses in the source code instead of in the `notes.txt`. In a way, debug printing forces you to tackle bugs scientifically because you have to write out hypotheses as print statements.

Extra Credit

- Find a graphical debugger and compare using it to raw `GDB`. These are useful when the program you're looking at is local, but they are pointless if you have to debug a program on a server.
- You can enable core dumps on your OS, and when a program

crashes, you'll get a core file. This core file is like a postmortem of the program that you can load up to see what happened right at the crash and what caused it. Change `ex18.c` so that it crashes after a few iterations, then try to get a core dump and analyze it.