# Lab 3: Introduction to Debugging

## Introduction to This Lab

Great news! You don't have to write flawless code to be a great programmer. Even the best programmers make mistakes, but they know how to use tools to fix them. Effective use of a debugger can really help solve bugs. As the name might suggest, the debugger does **not** actually debug your code for you, but rather aid you in the process.

We'll learn how to use the debugger to pause the execution of your program, run a line at a time, and inspect variables as they change. These techniques alone will help you solve (or at least identify) most any bug. You may have used a debugger built into an IDE (e.g. Eclipse) before. We will be doing the same thing, but from the command line.

## Goals

- Learn to load an executable into the debugger
- Use debugger commands to walk through and explore a program
- Identify the bug in a program

## Generating the Deliverable

For this lab, you will need to **turn in** a file that demonstrates some of the work you did. We'll use the `script` command to ask our shell (called `bash`) to log each command we type to a file. At the end of the lab, we will stop logging, and learn how to submit the file `script` generated.

1. Create a new directory `~/1730/lab03/` and change (`cd`) into it.
2. Type `script transcript.log` to begin logging your commands to a basic text file. You should see the message `Script started, file is transcript.log` as well as a new prompt.

> ⓘ **Note:** `script` is meant to keep a log of basic shell input and output. It doesn't handled interactive programs (i.e. they take up the entire terminal window) too well. Much of our work in this lab will be in the debugger or in your editor, both of which are interactive, so your `transcript.log` file will likely end up with garbage-like data, but you don't need to worry about that.

## Example Files

You will need some example files to work with. We'll use `wget` to download them, as before.

3. Use `wget` to download the following URL:
   `http://cs.uga.edu/~cs1730/lab03/buggy.cpp`

## Loading an Executable into the Debugger

The debugger we'll use in this class is called GDB (short for "GNU Debugger"). As with the g++ compiler (also GNU software), GDB is open source, free, and popular. GDB works only on executables (not source code), so if your code won't compile in the first place, you're left with only the compiler's error messages. In other words, debuggers are used to look for runtime errors or potentially logical errors. The compiler is responsible for catching syntax errors.

TIP: If you're stuck with a compile-time error, try your best to figure out what the error actually means. Search the web for tips. The error messages aren't always useful, but you should give them a chance!

> ✎ **Tip:** Don't ignore compiler *warnings*. Your code can compile with warnings (as opposed to *errors*), but many warnings exist to help prevent runtime errors.

While you can load any executable into the debugger, you won't be able to do much unless it was compiled for the debugger. Typically, compilers try to optimize the executable for performance and a small file size, so only essential information is stored. When you compile for the debugger, the compiler will include extra information (e.g. line numbers, original source code, variable names).

4. Take a look at buggy.cpp to see what we're up against. Use your editor or type `cat buggy.cpp`. It's a pretty simple program with an array representing a queue of numbers. The programmer decided that the front of the queue is at the right. Notice that 42 is at the front of the queue.

5. Compile and run the program and test it out.
   a. `g++ -o buggy buggy.cpp`
   b. `./buggy`
   c. Let's try and inspect the front of the line. When it asks, request position `0`. You would expect to see `42`, right? You should get some other number instead. Looks like a bug!

6. Compile `buggy.cpp` again, but this time for the debugger: just add the `-g` flag. Your command should look something like: `g++ -g -o buggy`

`buggy.cpp`. This executable will still run on the command line, but it has additional information inside of it.

> ✎ **Tip:** In future projects, add a `debug` target to your makefile so you don't have to type this in every time you want to debug! You could also modify the primary compile command, but you don't want the final product to have debugging info stored inside of it.

7. Now, load it into the debugger by typing `gdb buggy`. After a barrage of legal information, you should see a new prompt: `(gdb)` ! This is similar to your normal command prompt (i.e. shell), but it is a completely different program and only understands debugger commands.

8. Type `run` (shortcut: `r`) and press `enter`. Inspect position 0 again, when it asks. After the result gets printed, you'll see the message `Program exited normally.`. Sometimes the debugger will give us some hints when there is a problem, but it obviously isn't helping much in this scenario.

9. Run `break main` (shortcut: `b main`). This will *set a breakpoint* at the beginning of the main function, meaning that whenever `main` is called, execution will be paused and you will be presented with a GDB prompt. Once you set a breakpoint, it will stay set until you clear it, or you exit GDB. Notice that it reports back to you the associated file and line number of main.

10. Run `r` to execute the program again. It should immediately stop at the breakpoint you just set. It should also print out the next line to be executed (line 6; the first line in main).

11. Let's execute that first line then stop again. Run `next` (shortcut: `n`) to accomplish that. You'll see that the `cout` line of code (line 8) is printed indicating that it will be executed if you type `next` again. But wait a minute—what happened to line 7?

> ⓘ **Note:** `step` (shortcut: `s`) is an alternative to `next`. Whereas `next` will execute a function call all at once, `step` will actually go inside of a function call so you can walk through line-by-line. You won't need this since we don't have any function calls in this example, but it's worth remembering.

12. Run `list` (shortcut `l`) to see a snippet of code near the current location. Notice that line 7 is the declaration of a variable called `choice`. Read the following note to see why GDB skipped over it.

ⓘ **Note:** Not every line of code directly correlates to instructions sent to the processor. For example, blank lines are just ignored, as you might expect. Variable declarations, too—they're built into the containing function at compile time, so memory gets allocated as soon as that containing function is placed on the stack. Actually initializing the variables *does* require processor time though, so that's why GDB stopped at line 6.

13. You can print out the current value of a variable with `print`. Try `print queue[7]` (shortcut: `p queue[7]`). That should print out 42, which is what we expected, so the bug is likely later on in the code.

    You can even print out the entire array with `p queue`. Also notice how each variable you print out is assigned to a GDB variable (`$` and a number) for you to reference or access in later commands, should you want.

ⓘ **Note:** You can directly modify variables in GDB with the `set` and the equals sign. For example, feel free to `set queue[3]=123` (it won't mess up this lab).

14. Now let's take a programmatic approach to solving this bug. We can work backwards from the line that presents the bug, inspecting variables to see if they line up with our expectations. We won't need the breakpoint at main anymore, so let's clear it with `clear main` (shortcut: `cl main`).

    Let's type `l` to decide where to set our next breakpoint. Line 12 is where we get erroneous output, so let's stop right before that line is executed: `b 12`. Our program is still paused, so we need to tell it to start back up with `continue` (shortcut: `c`). Note that `run` would start the program over from the beginning again.

✎ **Tip:** If you just press `enter` at a blank prompt, the previous command will be run. This is handing for running one line at a time multiple times, or listing more lines of source code. We could have done this instead of setting a new breakpoint since our code is short.

15. The program will wait for your input when it gets to the `cin` line. Type in `0`, just like you did before to reproduce the bug. Next, you should hit the breakpoint and see that line 12 is about to be executed. Now, let's inspect `queue[index]`. Type `p queue[index]`. Still not 42, like we expected.

16. Now inspect the index variable. `p index`. Notice anything weird? Spoiler alert: the solution is next.

You should see `8` printed. Hopefully this raises some flags, since the `queue` array is of size 8. We should only access up to index 7! A bit more inspection reveals an off-by-one error on line 10.

17. To exit GDB, type `quit` or press `control`+`d`. Now try fixing the bug, recompiling, and trying again!

## Other Tips

- If you don't know what you expect the code to do in the first place, debugging won't be much help. Seek to understand every line first.
- If you don't know where the bug is, just pick somewhere (probably towards the beginning) to start. Verify that part works (by inspecting variables with the debugger), then move on. Eventually, something should fail to line up with what you expect.
- GDB is capable of much more than what we saw in this lab, but much of it requires a better understanding of how the language works. These basics will go a *long* way though!

## Submitting the Deliverable

18. End the `script` command by exiting the shell (e.g. type `exit`). You should see a message confirming that the script is done.

19. `cd` to the folder directly above `lab03` (`~/1730/`) and type `submit lab03 cs1730` to send in the deliverable for this lab.

## Conclusion

Hopefully, the value in knowing how to use a debugger is clear. There will still be some cases when other debug techniques will be more appropriate (including the simplistic “add more output” approach), but in many cases, the debugger will save you time, especially as your programs get more complex!