CS 211 Lab 4

Debugging

Winter 2021

Today we are going to practice debugging using *emacs* and *GDB*.

GDB is a debugger lets you see inside your program while it runs. You can step through it line by line, or you can choose a particular place to stop—a **breakpoint**—and then have it continue at full speed until it reaches the point where you've asked it to pause. You can print out the values of objects, modify their values, and even call functions.

Getting Started

The starter code is available at ~cs211/lab/lab04.tgz, so you can extract it into your current working directory with the command

```
% tar -xkvf ~cs211/lab/lab04.tgz
```

Your current working directory should now contain a subdirectory called lab04.

Revised Setup

We've also improved several things about the 211 environment setup to solve a couple of common student mistakes and ensure it doesn't mess with future classes you take. You should re-run the installer now:

```
% ~cs211/setup211
```

After you have done so, log out and back in.

Now, whenever you log in, instead of running *fish*, you will run a new command *211*, which will set up the environment and execute *fish* in a single step:

```
% 211
```

We'll remind you each time you log in with a message that says: "Run '211' for the CS 211 development environment".

Guide to Emacs and GDB

For this lab, we will provide you with some code that needs to be debugged. Afterwards, hopefully you see some value in using *GDB* on your own code when you are having problems.

To use the debugger, you need to compile your C code with the -g flag. The Makefiles we supply you with in each lab and homework comes with that enabled already, so you should be all set. But if you run *cc* by hand, or use a future project with a different build system, don't forget to pass it the -g flag.

GDB can be run directly from the shell, but if you run it in Emacs then Emacs will help you out by showing you the debugger's position in your code. You *definitely* want that, because the way *GDB* shows context on its own is not very friendly. But to use *GDB* from Emacs, you'll need to learn a bit more Emacs. In particular, you need to understand windows and buffers, and you need to be able to issue Emacs "***M-x***" commands.

Emacs Keys & Commands

Windows & Buffers

Emacs lets you edit more than one file at a time, and that can include files that are open but aren't currently displayed. Each thing you're working on (usually a file, but possibly a compiler run or *GDB* session), whether visible or not, is kept in a **buffer**. The rectangles in the terminal that some buffers are displayed in are called **windows**. You can open and close windows, switch between them, and switch which buffer is displayed in each window.

C-x 2 Split this window in 2.

C-x 1 Close all windows but this 1.

C-x 0 Close the current window. (That's a zero.)

C-x o Other window, go there.

C-x b View a different buffer in this window.

C-x C-b Show a list of buffers.

C-x C-f Find a file to open in this window.

Running Things

M-x introduces commands that are whole words (or Emacs Lisp function names) rather than just keystrokes. So when you press it, a command line (called the "minibuffer") will open at the bottom of Emacs for you to type the rest of the command (and press Enter).

Ah, but how do you type M(eta)-, a modifier key that your keyboard likely doesn't have? You can configure your terminal to send a key you do have as Meta—usually you'd map Alt on Windows or

Option on Mac. But if you haven't set that up, you can always type an ***M-k*** combination by typing Escape followed by the particular *k*.

So to type ***M-x***, you can press Escape followed by x, regardless of your particular hardware or configuration. ***Option-x*** or ***Alt-x*** can be made to work and might already.

M-x compile Run the compiler. (Emacs will ask what command to run. If you're currently editing a file in src/ or test/ then *make* won't find the Makefile in there; change the compilation command to `% cd .. && make -k` to *cd* up a level before running *make*.)

M-x recompile Run the compiler again with the same command.

M-x shell Open a shell in a buffer.

M-x gdb Start *GDB*. (The default command that Emacs suggests is good.)

Getting Unstuck

At some point, you may find yourself stuck in the minibuffer with Emacs complaining about every keystroke. To get out, use:

C-g Quit the current operation.

Help

C-h a Search for commands *apropos to* a word.

C-h d Search the documentation.

C-h k Get help on a key combination.

C-h l See your last 300 keystrokes (to find out what you did just now).

C-h ? Get help on help.

GDB Commands

help Displays help; follow it with a command name to get help on that command.

Enter Repeats the previous command again.

Finding & Loading

file *⟨FILE⟩* Tells GDB where to load your program from. This is relative to GDB's working directory, so something like `../count`.

pwd Prints GDB's working directory.

cd *⟨DIR⟩* Changes GDB's working directory.

Starting & Stopping

break *⟨point⟩* Sets a breakpoint at a function, a line number, or *⟨file⟩*:*⟨line⟩*).

clear *⟨point⟩* Deletes any breakpoints at *⟨point⟩*; omit *⟨point⟩* to clear a breakpoint on the current line.

run *⟨args⟩*... Runs your program from the start; optionally passes *⟨args⟩* as command-line arguments.

C-c C-c Pauses the running program. Code will stop on whatever line was executing when the key combination was pressed.

next Executes until the next line of your program, not looking inside function calls.

step Executes one small step of your program, including stepping into function calls.

finish Resumes your program for the remainder of the current function call.

continue Resumes your program from where it's paused and runs to the next breakpoint.

Peeking & Poking

print *⟨EXPR⟩* Prints the value of a variable in your program (in scope at the execution point), or the value of a larger expression in the context of your program. (Can even call functions!)

set variable *⟨VARNAME⟩* = *⟨EXPR⟩* Modifies the value of a variable in your program.

Debugging some code

First, enter the lab04/ directory and run *make*.

Fixing an infinite loop

The executable *infinite* is created from src/infinite.c Try compiling the code and running it first. As you may have guessed, it runs forever. This isn't what it's supposed to do, however. It has a bug. Let's use GDB to figure out where the unintentionally-infinite loop is.

Remember that you can use the command ***C-c*** in the shell to terminate a running program.

1. Open *emacs* without specifying any filename.
2. Start a *GDB* session with ***M-x gdb***. Then hit *Enter* to execute the command and open a multi-window layout. The top-left window is where you type commands. The top-right window displays local variables when code is paused. The middle-left window displays the code currently open.
3. In the top-left window, type ***pwd*** to see what directory *GDB* is running in.
4. In the top-left window, type ***file infinite*** to start debugging the *infinite* executable. Note that you have to specify the relative path to the executable you want to debug, so if ***pwd*** didn't show *GDB* running in lab04/ you will have to type the path to *infinite* instead.
5. In the top-left window, type ***run*** to start running the code. Just like when you run it outside of *GDB* this code will run forever without printing anything.
6. Press the key combination ***C-c C-c*** to pause the running code. The middle-left window will now display what line of code was running when execution was paused.
7. In the top-left window, type ***step*** to move through code line-by-line. As you do, you will see the current line changing in the middle-left window and the local variables and their values changing in the top-right window.

You can hit *Enter* in *GDB* to run the previous command again. This is really useful when calling ***step*** multiple times in a row!

8. You can also use the ***print*** command to display the value of variables. For example, you can type ***print s*** to display the input argument to the function.
9. Step through the code for a while, watching the local variable values, and see if you can figure out why this code runs infinitely.
10. To close the *GDB* session, you can type ***quit*** in the top-left window and then press ***C-x C-c*** to exit *emacs*.

Understanding nonfunctional code

Broken code doesn't always infinite loop. Sometimes it completes quickly, but doesn't provide a correct result. Let's use *GDB* to take a look at a second program, *broken*, that runs but doesn't work properly.

First, try running *broken* in the shell and take a look at the code for it in src/broken.c. It currently prints out the wrong summation for the array. It does finish running though, so this time we'll have to debug it using *breakpoints*.

1. The first few steps are the same as last time: open *emacs*, start a *GDB* session, and use the ***file*** command to specify which executable you are debugging (this time it should be *broken*).

2. In the top-left window, type ***break sum_array***. This will create a breakpoint at the start of that function, any time it is called.

3. In the top-left window, type ***run*** to start running the code. This time, execution will pause automatically once the breakpoint is reached.

4. Like last time, use ***step*** and the local variable values in the top-right window to determine what is going wrong and fix it.