CS 211 Lab 4

Debugging

Winter 2022

Today we are going to practice debugging using *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`.

Guide to 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 `Makefile`s 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.

By default *GDB* isn't very easy to understand. The update we made to your *GDB* configuration however, improves it quite a bit. It will show you which line of code is executing as well as the current values of every local variable. It actually shows you far more information than you need, but you can limit what *GDB* displays, as we'll instruct you to do below.

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.

Ctrl-c Pauses the running program. Code will stop on whatever line was executing when the key combination was pressed. **Warning:** only press Ctrl-c once. Pressing it multiple times freezes *GDB* for some reason.

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.

Changing Dashboards

dashboard -layout breakpoints source variables Reduces the number of output displays to remove information that's irrelevant for this class. Pressing tab while typing the command lists other possible display output that could be enabled by adding it to the list, but those three should be enough.

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 `Ctrl-c` in the shell to terminate a running program.

1. Run *GDB* on the program with the command `gdb infinite`.
2. *GDB* should start and give you a prompt. Type `run` to start running the code. Just like when you run it outside of *GDB* this code will run forever without printing anything.
3. Press the key combination `Ctrl-c` (only once!) to pause the running code. A number of displays will appear with information about your code.
4. Many of these displays aren't all that useful. Type `dashboard -layout breakpoints source variables` to reduce the number of displays to ones with information we actually care about.

   Now there should be three or four displays. The top is possibly "Output/messages" which displays various output information including error messages from your code, but only displays when it has something to say.

   Below that is "Breakpoints" which lists any active break points in your code (there shouldn't be any yet).

   Below that is "Source" which displays the line of code where execution was paused and several lines before/after it. This is probably displaying your code for *make_uppercase*(), but could also be displaying the source code for *toupper*() depending on where it paused.

   Finally, the "Variables" display lists all variables and their current values. It does its best to show multiple representations of variables: such as the decimal or character version, or both the memory address and contents of pointers.

5. Type step to move through code line-by-line. You can type it again, or use up arrow to retrieve recent commands and hit enter to run it again. As you do, you will see the current line changing in the "Source" display and the local variables and their values changing in the "Variables" display.

6. 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.

7. Step through the code for a while, watching the local variable values, and see if you can figure out why this code runs infinitely.

8. To close the *GDB* session, you can type quit and then answer y to the prompt to exit.

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: run gdb broken to start a *GDB* session debugging the executable.

2. You should limit the default displays again by typing dashboard -layout breakpoints source variables. You can do this right away rather than waiting for the displays to appear first.

3. Now type break sum_array. This will create a breakpoint at the start of that function, any time it is called.

4. Type run to start running the code. This time, execution will pause automatically once the breakpoint is reached.

5. Like last time, use step and the local variable values in the "Variables" display to determine what is going wrong and fix it.

6. To close the *GDB* session, you can type quit and then answer y to the prompt to exit.