ECE220: Computer Systems & Programming **Fall 2022 ZJUI**
Lab 8 **Tuesday 15 November**

Learning to Use a Debugger

In today's lab, you will learn how to use a debugger (the Gnu debugger, GDB) to examine a C program. Debuggers help to isolate, identify, and understand buggy behavior more easily than is possible using only execution and additional printing. For complex programs, availability of a debugger makes testing much, much easier. In MP7, you must use GDB to explore several buggy programs, identify the problems, explain them in English, and, in some cases, fix the problems.

If you have used the command-line version of the LC-3 simulator, you have already used a similar interface to that provided by GDB. In fact, I based the commands in the LC-3 simulator on the commands available in GDB.

Begin by checking out the **`lab8`** subdirectory in your Subversion repository. The directory contains a copy of this document (**`lab8.pdf`**) and a C source file, **`factorial.c`**. The program contains a bug, which you can find and fix using GDB.

The Task

To use GDB effectively, you need to enable the built-in debugging support for the compiled executable. To do so, include the –g option when compiling, as with the factorial code in this week's Subversion directory:

```
gcc –g –Wall factorial.c –o fact
```

Run the program a couple of times to see how it behaves. You can start GDB by typing:

```
gdb
```

You can load the executable to debug by using the file command (the first part, "**`(gdb)`**", is the GDB prompt:

```
(gdb) file fact
```

Alternatively, you can pass the executable name as command-line argument when starting:

```
gdb fact
```

GDB has an interactive shell, much like the one you use when you log into the Linux machines. It can recall history with the arrow keys, auto-complete words (most of the time) with the TAB key, and has other nice features.

If you're ever confused about a command or just want more information, use the "**`help`**" command, with or without an argument:

```
(gdb) help [command]
```

To run the program, type:

```
(gdb) run
```

Or, if the program needs command line arguments:

```
(gdb) run [arg1] [arg2] …
```

The "**run**" command starts program execution. If the program has no serious problems, the program should run fine under GDB, too. If the program crashes, GDB can usually tell you some useful information, such as the line number at which the program crashed and parameters to the function that caused the error.

However you don't need to use GDB to run the program when the program is working. When the program isn't working, you probably want to pause execution and check the current status of program. To pause the program, you can use breakpoints.

Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command "**break**". This sets a breakpoint at a specified file-line pair:

```
(gdb) break factorial.c:22
```

This sets a breakpoint at line 22 of file **factorial.c**. You can set multiple breakpoints. The execution stops whenever it reaches any breakpoint.

You can look at your source code using the "**list**" command in GDB. You can list functions by name, for example:

```
(gdb) list factorial
```

If you just type "**list**" after starting GDB, the debugger will show you the start of main. "**List**" without arguments moves further along in the file. GDB remembers the last file that you examined, so if you see a line in the code you're looking at, you can omit the file name when setting a breakpoint.

Back to breakpoints. You can also tell GDB to stop when execution reaches a particular function:

```
(gdb) break factorial
```

The execution stops whenever the factorial function is called.

You can also stop execution only when a particular requirement (or a set of requirements) is satisfied. Using conditional breakpoints allows us to accomplish this goal. Conditional breakpoints are identical to regular breakpoints, except that you get to specify some criterion that must be met for the breakpoint to trigger:

```
(gdb) break factorial if n == 2
```

You can add a condition to an existing breakpoint using cond. Each breakpoint has a number, which GDB prints when you create the breakpoint. Let's say that the breakpoint in function factorial was #2. In that case, the above command is equivalent to:

```
(gdb) cond 2 n == 2
```

You can also change the condition on a breakpoint, or disable or enable each breakpoint after hitting it. You can also tell GDB to skip the first N times a breakpoint is hit. Use "**help**" to learn more—these variants are not used as frequently as basic breakpoints.

To resume execution, you can use the "**continue**" command:

```
(gdb) continue
```

The execution will continue until the next breakpoint, unless a fatal error occurs before reaching that point.

You can also single-step (execute just the next line of code) with the "**step**" command:

```
(gdb) step
```

Similar to "**step**," the "**next**" command executes the next line, but does not enter the source of any functions. Instead, "**next**" silently finishes execution of any called instructions, treating the whole line as one instruction.

```
(gdb) next
```

You may need to repeat the "**continue**," "**step**." or "**next**" commands multiple times. In GDB, you repeat the last command by just pressing ENTER.

When executing through loops, you may want to continue execution until the loop terminates. To do so, use the "**until**" command:

```
(gdb) until
```

Similarly, you may want to continue execution until the current function returns:

```
(gdb) finish
```

So far, you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line-by-line. When the execution pauses, you may want to see values of variables or even modify variables' values. You can do so with the "**print**" and "**set**" commands:

```
(gdb) print n
(gdb) set n = 2
```

The "**set**" command sometimes requires you to specify that you want to change a variable: "**set var n = 2**". You can call functions in your program as part of a "**print**" command. GDB uses the stack provided by the hardware and executes the called function in your program. Be careful: those functions can also crash if they are given bad values. If you just want to call a function rather than printing or calculating something based on its return value, use the "**call**" command instead. Sometimes, it's useful to add functions to your program solely for use from within GDB.

If you want to print an expression or a variable's value every time GDB stops execution, use the "**display**" command instead. This command fails when variables in the expression go out of scope, and may change meaning if variables with the same names come into scope (remember that the program has a current point of execution that defines which variables are in scope—that is, usable by name).

You can use the "**backtrace**" command to view a trace of the function calls that brought execution to the current point (the current function, the function that called it, and so forth, all the way back to main). The "**backtrace**" command (also known as "**where**") is particularly useful when debugging with seg faults (illegal memory accesses, as sometimes arise from array bounds errors).

```
(gdb) backtrace
```

You can also use the "**up**" and "**down**" commands to change the current scope to the caller/callee function. You can specify how many levels you want to change with these two commands:

```
(gdb) up 2
(gdb) down 1
```

Other useful commands include "**info**," "**delete**," and "**clear**."

You can use the "**info**" command to list the current breakpoints ("**info breakpoints**"), local variables ("**info locals**"), file scope/global variables ("**info variables**") register values ("**info registers**"), and so forth.

You can use the "**delete**" command to delete a breakpoint (or disable it temporarily with the "**disable**" command, then re-enable it later with the "**enable**" command).

```
(gdb) delete breakpoint 2
```

You can use the "**clear**" command to delete all breakpoints set inside a function.

```
(gdb) clear factorial
```

Play with GDB for a while and learn to use the various commands explained here, then find the bug in the factorial program and fix it.

If you have extra time, you may wish to take a look at MP7 and get started debugging the problems given to you in that assignment, but remember that the MP is individual work, whereas you can discuss this tutorial material freely.