Debugging with GDB

Quick Links: **Home (https://q.utoronto.ca/courses/419441/assignments/syllabus)** **Lecture material (https://q.utoronto.ca/courses/419441/pages/lecture-material)** **Lab assignments (https://q.utoronto.ca/courses/419441/pages/lab-assignments)** **Piazza Discussion (https://piazza.com/utoronto.ca/winter2026/ece353/home)** This page brought to you by Prof. Ashvin Goel.

What is GDB and why do you need to use it?

GDB is a debugger. Its purpose is to help you analyze the behavior of your program, and thus help you diagnose bugs or mistakes. It will be indispensable when working on your labs. With GDB you can do the following things:

- Control aspects of the environment that your program will run in.
- Start your program, or connect up to an already-started copy.
- Make your program stop for inspection or under specified conditions.
- Step through your program one line at a time, or one machine instruction at a time.
- Inspect the state of your program once it has stopped.
- Change the state of your program and then allow it to resume execution.

In your previous programming experience, you may have managed without using a debugger. You might have been able to find the mistakes in your programs by printing things on the screen or simply reading through your code. Beware, however, that the OS labs are complicated, making the debugger an essential tool in this environment. You should, therefore, take the time to learn GDB and make it your best friend. This guide will explain to you the most common GDB commands, and suggest some helpful debugging techniques.

How to start debugging your code

You can debug your program by running `gdb` on your program. However, before you do so, make sure that the program is compiled with the "`-g`" option to `gcc`.

```
gdb program-name
```

Most common GDB commands

l, list - List lines from source files.

Use this command to display parts of your source file. For example, typing

```
(gdb) l 101
```

will display line 101 in your source file. If you have more than one source file, precede the line number by the file name and a colon:

```
(gdb) l thread.c:101
```

Instead of specifying a line number, you can give a function name, in which case the listing will begin at the top of that function.

b, break - set a breakpoint.

Use this command to specify that your program should stop execution at a certain line or function. Typing

```
(gdb) b 18
```

means that your program will stop every time it executes a statement on line 18. As with the "list" command, you can specify to break at a function, e.g.:

```
(gdb) b main
```

d, delete - Delete breakpoint (or other things).

Use this command to delete a breakpoint. By typing

```
(gdb) d 1
```

you will delete the breakpoint number "1". GDB displays the number of a breakpoint when you set that breakpoint. Typing "d" without arguments will cause the deletion of all breakpoints.

clear - Clear a breakpoint.

If you don't remember the number of the breakpoint you want to delete, use the "clear" command. Just like the "breakpoint" command, it takes a line number or a function name as an argument.

c, continue - continue execution.

After your program has stopped at a breakpoint, type

```
(gdb) c
```

if you want your program to continue the execution until the next breakpoint.

s, step - Step through the program. If you want to go through your program step by step after it has hit a breakpoint, use the "step" command. Typing

```
(gdb) s
```

will execute the next line of code. If the next line is a function call, the debugger will step into this function.

n, next - Execute the next line.

This command is similar to the "step" command, except for it does not step into a function, but executes it, as if it were a simple statement.

disable, enable - Disable/enable a breakpoint.

Use these commands with a breakpoint number as an argument to disable or enable a breakpoint.

display - Display an expression. Display a value of an expression every time the program stops. Typing

```
(gdb) display x
```

Will print the value of a variable "x" every time the program hits a breakpoint. If you want to print the value in hex, type:

```
(gdb) display /x x
```

undisp - Cancel the display of some expressions.

Arguments are the code numbers of the expressions. If no arguments are given, GDB will cancel all expression displays.

print, printf - Print an expression.

To print a value of an expression only once, use the "print" command. It takes the same arguments as the "display" command.

The "printf" command allows you to specify the formatting of the output, just like you do with a C library printf() function. For example, you can type:

```
(gdb) printf "X = %d, Y = %d\n",X,Y
```

command - Execute a command on a breakpoint.

You can specify that a certain command, or a number of commands be executed at a breakpoint. For example, to specify that a certain string and a certain value are printed every time you stop at breakpoint 2, you could type:

```
(gdb) command 2
 > printf "theString = %s\n", theString
 > print /x x
 > end
```

bt, where - Display backtrace.

To find out where you are in the execution of your program and how you got there, use one of these commands. This will show the backtrace of the execution, including the function names and arguments.

set - Assign a value to a variable.

Sometimes it is useful to change the value of a variable while the program is running. For example if you have a variable "x", typing

```
(gdb) set variable x = 15
```

will change the value of "x" to 15.

Then you could invoke it just by typing "db". (If you put this or other commands in a file called .gdbinit, GDB will execute them automatically at startup time.)

info - Display information.

With this command you can get information about various things in your debugging session. For example, to list all breakpoints, type:

```
(gdb) info breakpoints
```

To see the current state of the hardware machine registers, type:

```
(gdb) info registers
```

help - Get help (on gdb, not your labs!)

Finally, if you want to find more about a particular command just type:

```
(gdb) help "command name"
```

kill - You can also tell GDB to kill the process it's debugging.

This will cause your program to exit unceremoniously, much as if you'd gone to its window and typed ^C:

```
(gdb) kill
```

Debugging tips

Tip #1: Check your beliefs about the program

So how do you actually approach debugging? When you have a bug in a program, it means that you have a particular belief about how your program should behave, and somewhere in the program this belief is violated. For example, you may believe that a certain variable should always be 0 when you start a "for" loop, or a particular pointer can never be NULL in a certain "if statement". To check such beliefs, set a breakpoint in the debugger at a line where you can check the validity of your belief. And when your program hits the breakpoint, ask the debugger to display the value of the variable in question.

Tip #2: Narrow down your search

If you have a situation where a variable does not have the value you expect, and you want to find a place where it is modified, instead of walking through the entire program line by line, you can check the value of the variable at several points in the program and narrow down the location of the misbehaving code.

Tip #3: Walk through your code

Steve Maguire (the author of Writing Solid Code) recommends using the debugger to step through every new line of code you write, at least once, in order to understand exactly what your code is doing. It helps you visually verify that your program is behaving more or less as intended. With judicious use, the step, next and finish commands can help you trace through complex code quickly and make it possible to examine key data structures as they are built.

Tip #4: Use good tools

Using GDB with a visual front-end can be very helpful. For example, using GDB inside the emacs editor puts you in a split-window mode, where in one of the windows you run your GDB session, and in the other window the GDB moves an arrow through the lines of your source file as they are executed. To use GDB through emacs do the following:

1. Start emacs.
2. Type the "meta" key followed by an "x".
3. At the prompt type "gdb". Emacs will display the message:
   ```
   Run gdb (like this): gdb
   ```

4. Add the program name, so you should have:

   ```
   Run gdb (like this): gdb program
   ```

   displayed in the control window.

At this point you can continue using GDB as explained above.

Tip #5: Beware of printfs!

A lot of programmers like to find mistakes in their programs by inserting "printf" statements that display the values of the variables. If you decide to resort to this technique, you have to keep in mind two things: First, because adding printfs requires a recompile, printf debugging may take longer overall than using a debugger.

More subtly, if you are debugging a multi-threaded program, the order in which the instructions are executed depends on how your threads are scheduled, and some bugs may or may not manifest themselves under a particular execution scenario.

Where to go for help

For help on GDB commands, type "help" from inside GDB. You can find the **documentation on GDB here (http://www.gnu.org/software/gdb/gdb.html)**. And of course your friendly TAs are always there to help!