Brief GDB Tutorial

Claudio A. Parra

University of California

parraca at uci dot edu

October 9, 2021

Overview

1. The Basics
   - what is gdb
   - getting started
   - run
2. Stopping the Execution
   - break and continue
   - conditional break
   - next, step, and finish
3. Viewing Data
   - print
   - dump memory content
   - watch
   - other commands

What is GDB?

- `gdb` stands for **G**NU **De**bugger.
- A debugger is a program that helps you analyze the execution of another program. GDB is the *de-facto* debugging tool in Linux. If you are a Mac user, use `lldb`.
- Some tasks you can perform in `gdb` are:
  - Execute one line of code at the time.
  - Run the code until a given point.
  - Stop the execution based on conditions.
  - Stop the execution based on a variable being modified.
  - See the current content of variables, registers, and execution stack.

Getting started

In order to add debugging info to the compiled code, add the flag `-g` to the compiler.

Terminal

```
// gcc <compiler flags> -g <source code> <linker flags>
$ gcc -Wall -Werror -o my_prog.bin -g my_prog.c -lm
```

Now `my_prog.bin` contains information about its own source code, line count, variable and function names, etc.

To start gdb, just type it and follow it by the name of the binary you want to analyze.

Terminal

```
$ gdb my_prog.bin
(gdb)
```

Getting started II

GDB works similarly to your system's shell:

- Type commands, and press enter.
- Press ↑ or ↓ to see the history.
- Press ⭾Tab to auto-complete (most of the time).
- While on it, you can call `help`, or `help <command>` to get help about a particular command.
- If press just ↵, gdb runs the last command again.

run

It simply runs your program. If your program is correct, then it should run flawlessly here too.

Terminal

```
(gdb) run
Starting program:  /home/user/.../my_prog.bin
// all the printing of your code

[Inferior 1 (process <PID>) exited normally]
(gdb)
```

run

Now, this does not mean that the **logic** of your code is correct, or that there are no cases in which your code breaks. If you get a serious error, like a segmentation fault, gdb will give you some useful information too.

Terminal

```
(gdb) run
Starting program:  /home/user/.../my_prog.bin
// all the printing of your code

Program received signal SIGSEGV, Segmentation fault.
0x000055555555478f in main () at my_prog.c:27
27        d->x = i;
(gdb)
```

Here we can see that line 27 triggered the segfault. If you are in this situation, probably you want to slow down things and see what is happening.

`break` and `continue`

The `break` command establishes a "breakpoint" lets you stop the execution of your code if the execution ever reaches that point:

- A particular line of code `(gdb) break my_prog.c:26`
- At a particular function `(gdb) break get_distance`

Once you have setup all the breakpoints that you want, you may start the program from the beginning with `run`. If you reach a breakpoint, use the command `continue` to continue the execution. The code will stop in the next breakpoint (if there is any).

With `info breakpoints` you can see a list of your breakpoints. With `delete <b#>` you can delete a breakpoint, where <b#> is its number.

break and continue

Terminal

```
(gdb) break my_prog.c:26
Breakpoint 1 at 0x79a:  file my_prog.c, line 26.
(gdb) break get_distance
Breakpoint 2 at 0x70a:  file my_prog.c, line 12.
(gdb) info breakpoints
Num  Type        Disp  Enb  Address             What
1    breakpoint  keep  y    0x00000000000007a0  in main at my_prog.c:26
2    breakpoint  keep  y    0x000000000000070a  in get_distance
                                                at my_prog.c:12
(gdb) run
Starting program:  /home/user/.../my_prog.bin


Breakpoint 1, main () at my_prog.c:26
26          for(i=0; i<n_dots; i++){
(gdb) delete 2
(gdb) info breakpoints
Num  Type        Disp  Enb  Address             What
1    breakpoint  keep  y    0x00000000000007a0  in main at my_prog.c:26
     breakpoint already hit 1 time
(gdb)
```

break <where> if <condition>

Many times you want to stop the execution only if certain condition is true. In those cases you can use a conditional `break` that stops only if the condition is true:

Terminal

```
(gdb) break my_prog.c:36 if i == n_dots - 1
Breakpoint 1 at 0x805:  file my_prog.c, line 36.
(gdb) run
// all the print of your code
Breakpoint 1, main () at my_prog.c:36
(gdb) print d
36          dis = get_distance(d, d->next);
(gdb) print i
$1 = 15
(gdb) print n_dots
$2 = 16
(gdb)
```

`next`, `step` and `finish`

If you want to run one line of code at the time, then you want to use this commands. **The difference is that `next` treats function calls as one line, but `step` gets into the function.** `finish` finishes executing the current function.

Terminal

```
(gdb) break my_prog.c:35
Breakpoint 1 at 0x7fc:  file my_prog.c, line 35.
(gdb) run
Starting program:  /home/user/.../my_prog.bin
Breakpoint 1, main () at my_prog.c:35
35          for(i=0; i<n_dots; i++){
(gdb) step
36          dis = get_distance(d, d->next);
(gdb) step
get_distance (p=0x555555756260, q=0x555555756280) at my_prog.c:12
12          int dx = q->x - p->x;
(gdb) step
13          int dy = q->y - p->y;
(gdb)
```

`next`, `step` and `finish`

If you want to run one line of code at the time, then you want to use this commands. **The difference is that `next` treats function calls as one line, but `step` gets into the function**. `finish` finishes executing the current function.

Terminal

```
(gdb) break my_prog.c:35
Breakpoint 1 at 0x7fc:  file my_prog.c, line 35.
(gdb) run
Starting program:  /home/user/.../my_prog.bin
Breakpoint 1, main () at my_prog.c:35
35          for(i=0; i<n_dots; i++){
(gdb) next
36          dis = get_distance(d, d->next);
(gdb) next
39          d->next->x, d->next->y);
(gdb) next
37          printf("(%d,%d)--- %2f ---(%d,%d)\n",
(gdb)
```

print

The `print` command is very versatile. You can print different kinds of variables:

Terminal

```
(gdb) print n_dots
$1 = 16 // integer in decimal
(gdb) print/x n_dots
$2 = 0x10 // integer in hexadecimal
(gdb) print d
$3 = (struct Point *) 0x555555756260 // pointers
(gdb) print *(d)
$4 = {x = 0, y = 0, next = 0x0} // structures
(gdb) print $sp
$1 = (void *) 0x7fffffffdc70 // registers
(gdb)
```

`x/<num><fmt> <obj>`

This command, similar to print, let you see the content of memory addresses or registers.

`<obj>` : the address or register (if register, precede with a `$`).

`<num>` : the number of elements to print from `<obj>`.

`<fmt>` : format to use: `x`, `d`, `f`, `c`...

Terminal

```
(gdb) x/8x $sp // print 8 elements in hex starting from $s
0x7fffffffdc70:  0x3b40 0xf7de 0x7fff 0x0000 0x0000 0x0000 0x0000 0x0000
(gdb)
```

`watch`

The command `watch` allows you to stop the execution whenever a particular variable is modified. The variable must be in the current execution scope:

Terminal

```
(gdb) watch d
Hardware watchpoint 2:  d
(gdb) continue
Continuing.
Hardware watchpoint 2:  d
Old value = (struct Point *) 0x555555756260
New value = (struct Point *) 0x555555756280
main () at my_prog.c:26
26          for(i=0; i<n_dots; i++){
(gdb)
```

Other Useful Commands

- `list` : prints the source code surrounding the current line.
- `where` : prints the execution call stack at the current point.
- `backtrace` : similar to `where`, but after a crash.
- `info frame` : shows list of CPU registers that compose the current stack frame.
- `info reg` : shows list of CPU registers, and their values.