GDB Tutorial

Pedro Abreu

August 26, 2019

GDB stands for "GNU Debugger", and it is a tool that gives you the power to inspect your program while it is executing. It allows you to freeze the execution and check the variable values, check if a particular condition is met, if a function was called, and a lot of other useful perks that help you to find what is going wrong with your program.

As you may have already noticed, pointers can be quite tricky to debug. Hopefully this tutorial will make your life a lot easier from here on.

To follow this tutorial clone the `gdb-tutorial` directory just like any other homework

```
$ git clone https://github.com/pedrotst/gdb-tutorial
```

Navigate to the `src` folder and you should see the following `main.c`:

```
#include <stdlib.h>
#include <stdio.h>
#include "my_list.h"

int main(){
    my_list *l = NULL;

    l = insert_node(1, l);
    l = insert_node(2, l);
    l = insert_node(3, l);
    l = insert_node(4, l);

    print_list(l);
    delete_list(l);

    return 0;
}
```

This program uses the provided `my_list` implementation to create a singly-linked list. This list should have four elements. Let's try to run this code and see what happens...

Start by compiling `my_list.c` to an object file:

```
$ gcc -Wall -Werror -std=c99 -c my_list.c
```

Now compile and run the `main` executable:

```
$ gcc -Wall -Werror -std=c99 main.c my_list.o -o main
$ ./main
Segmentation fault
```

Uh-oh, `Segmentation fault`! And conveniently, it doesn't even tell us where the program failed. There are a number of things we can do at this point. We could sprinkle `printf()`s and `fflush()`s all over, for instance. Another solution, unsuprisingly at this point, is GDB!

In order to use GDB, we must first take a step back and recompile our code with the `-g` flag. This creates an object file that contains "debug symbols". These symbols are leveraged by GDB to provide more detailed and useful information to the programmer.

```
$ gcc -Wall -Werror -std=c99 -g -c my_list.c
$ gcc -Wall -Werror -std=c99 -g main.c my_list.o -o main
```

Note: when you are implementing the homeworks, you don't have to worry about the `-g` *flag because it is already included in the* `Makefile`*. It is worthwhile, however, to rememeber this for your future endeavors.*

With the files compiled with debug symbols, we have everything we need to use GDB:

```
$ gdb main
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main...done.
(gdb)
```

Alright! `Reading symbols from main...done.` means that GDB was able to successfully read the debug symbols that we included in the previous step.

We can now run the program...

```
(gdb) run
Starting program: /u/antor/u7/$USER/cs240-hws/gdb-tutorial/example/main
Program received signal SIGSEGV, Segmentation fault.
0x00005555555548f4 in insert_node (val=2, l=0x555555756260) at my_list.c:31
31          last->next = create_node(val);
```

This is quite useful! GDB is able to provide us with the exact location in the our code where the crash occurred. We can see this is on line 31 inside `my_list.c`. This is part of the `insert_node()` function. line marked in red means that the program crashed

Let's set a breakpoint on this line. This will cause the program to hault *before* executing the code on line 31...

```
(gdb) break my_list.c:31
Breakpoint 1 at 0x8e3: file my_list.c, line 31.
```

Now we run the program again...

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /u/antor/u7/pdacost/cs240-hws/gdb-tutorial/example/main

Breakpoint 1, insert_node (val=2, l=0x555555756260) at my_list.c:31
31    last->next = create_node(val);
```

Did you notice that we used `r` *instead of* `run`*? This is because gdb allows us to be lazy and type only the first letter or two of a command. This means we can also use* `b`*,* `p`*,* `w`*, for break, print and watch respectively!*

Some notes on breakpoints. First, it is possible to set multiple breakpoints. You can also break on function names instead of line numbers. To view the current list of breakpoints, you can use the following command:

```
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000401299 in insert_node
                                                   at my_list.c:31
```

This is short for "info breakpoints". To remove a breakpoint, we can run `d 1` where 1 is the number. "d" is short for "delete."

Let's take a look on the code surrounding line.

```
(gdb) wh
```

You can also do this directly on launch, by running `gdb -tui main` *instead.*

This command should cause gdb to display the code surrounding the line in question. To close this view, press *Ctrl-C a.*

Alternatively, you can also use the `list` command:

```
(gdb) list
```

We can also now inspect the value of the variables:

```
(gdb) print last
$1 = (my_list *) 0x0
```

This means `last` has type `my_list *` and has value `0x0`, which of course is NULL. So when we run line 31 it will try to dereference a NULL pointer! This is the cause of our crash.

Let's execute the next line anyway:

```
(gdb) next
Program terminated with signal SIGSEGV, Segmentation fault.
```

Take a minute to open `my_list.c` and fix the bug. Once fixed, you should should see the following output:

```
$ main
[1, 2, 3, 4]
```

Don't forget to recompile your code when you are done fixing the issue(s)!

Already with just these few commands, you will be able to debug many programs.

To wrap up, here is a list of some other commands you may find useful in the future:

- `step`: Execute next program line (after stopping); step into any function calls in the line;
- `watch`: Every time a given variable changes it's value the program stops;
- `continue`: Continue running your program (after stopping, e.g. at a breakpoint);
- `clear`: Clears breakpoints;
- `backtrace`: Shows the backtrace of your program;
- `wh`: Shows a window with the surrounding code being executed.

For a list of all the commands run

```
(gdb) help
```

Remember: The best way to learn how to program is by sitting down and programming yourself! There is no magic.