CS382 Computer Organization and Architecture Fall 2021

Lab 4 · Debugging a C Program Using `gdb`

Lecturer: Philippos Mordohai, Shudong Hao *Date: September 22, 2021*

1 Objective

In this lab, we are going to get familiar with GNU debugger (`gdb`), which allows us to set breakpoints and then inspect the internal states of a program.

`gdb` can be used to debug C program, which is the main topic of this lab. After writing assembly, we can also use `gdb` to debug assembly programs, and it'll be the content of another lab in the future.

2 Installation

Generally Unix distributions come with `gdb`. You can also install it using the following commands from terminal:

```
sudo apt-get update
sudo apt-get install gdb
```

For more details refer to http://www.gdbtutorial.com/tutorial/how-install-gdb.

If you're using macOS with M1 chip, you can use `lldb` instead of `gdb`, because the latter doesn't support M1 chip yet. If you're using macOS with Intel chip, you might want to follow the instruction provided on Canvas. [1]

3 Debugging

In general you might be using the following command to compile your code:

```
gcc prog.cpp -o a.out
```

Now to enable your program to run with the debugger you need to put `-g` option. So the command becomes:

[1] From https://gist.github.com/danisfermi.

```
gcc -g prog.cpp -o a.out
```

You can start the debugger on your program by one of the following two ways. You can type the following command

```
gdb a.out
```

You can also enter the debugger first by using

```
gdb
```

and then type the name of the executable `a.out`.

Let's get familiar with a few commands in `gdb` for running your code. Try some of these out, but note that some won't work yet because they rely on other `gdb` commands being run first.

```
help command        # ask gdb to tell you about the named gdb command
run [arg1 ... argn] # run your code (and add command line arguments if you
 ↪have any)
```

Once you gave the `run` command, you'll notice that your program has been executed from the start to the end. This looks useless because it's the same as we run our program from terminal directly. The advantage of a debugger is that we can set a **breakpoint** somewhere in our code, so that the program will pause there and so we can see what's going on in the program.

3.1 `break`

We need the debugger to pause at certain point in the code so that we can investigate the program. To set a breakpoint we will use the command `break`.

```
break prog.c:12
```

In this example we are setting a breakpoint in file `prog.c` at line number 12. If we are interested in a particular function we can set break point at that function and the debugger will pause every time the function is called:

```
break function_name
```

Once you have set breakpoints, you can give the `run` command, and you'll see this time the program is paused at the breakpoints.

3.2 `continue`

To move on to next break point we can use the command `continue`, or simply `c`.

3.3 `step` and `next`

To proceed by single-step you can either use `step` or `next`, but there is a subtle difference between `step` and `next`. If your next line of code is a function call, `next` will consider it as a single instruction and will execute the function all at once. On the other hand, `step` will take you though lines of the function. So `step` gives more fine-grained control than `next`.

Also, if you suspect that the function has something to do with the error or bug, you might want to `step` into the function. But if you're sure the function is correct, then you can hit `next` which will run the function and bring you to the statement after the function call.

3.4 `print` and `display`

To print a value of a variable we can use `print` command.

```
print var   # var is a variable;
print *ptr  # ptr is a pointer.
```

The command `print` will print the value only once. If you want to print the value each time your are in the scope where the variable is defined you can use the command `display`.

3.5 `watch`

Whereas breakpoints interrupt the program at a particular line or function, *watch points* act on variables. They pause the program whenever a watched variable's value is modified.

```
watch var
```

Note: each time you make any change in the code you need to stop the `gdb` and recompile the program to generate the updated executable (`.out`). Then run the `gdb` with this new executable file.

4 Lab Task

Download the `gdb.zip` provided. The main function is declared in the file `test_int_array.c`, and the other two files have declaration and definition of an array data structure and its functionality.

This program uses dynamic allocation to create an integer array. It fills in the first half of the array with even numbers from 0 to 24, and the second half with odd numbers from 1 to 23. It then checks if some numbers are in the array, delete some numbers from the array, and then destroy the array in the end. When functioning properly, the code should have the following printed out on your terminal:

```
0
0 2
0 2 4
0 2 4 6
0 2 4 6 8
0 2 4 6 8 10
0 2 4 6 8 10 12
0 2 4 6 8 10 12 14
0 2 4 6 8 10 12 14 16
0 2 4 6 8 10 12 14 16 18
0 2 4 6 8 10 12 14 16 18 20
0 2 4 6 8 10 12 14 16 18 20 22
0 2 4 6 8 10 12 14 16 18 20 22 24
0 2 4 6 8 10 12 14 16 18 20 22 24 1
0 2 4 6 8 10 12 14 16 18 20 22 24 1 3
0 2 4 6 8 10 12 14 16 18 20 22 24 1 3 5
0 2 4 6 8 10 12 14 16 18 20 22 24 1 3 5 7
0 2 4 6 8 10 12 14 16 18 20 22 24 1 3 5 7 9
0 2 4 6 8 10 12 14 16 18 20 22 24 1 3 5 7 9 11
0 2 4 6 8 10 12 14 16 18 20 22 24 1 3 5 7 9 11 13
0 2 4 6 8 10 12 14 16 18 20 22 24 1 3 5 7 9 11 13 15
0 2 4 6 8 10 12 14 16 18 20 22 24 1 3 5 7 9 11 13 15 17
0 2 4 6 8 10 12 14 16 18 20 22 24 1 3 5 7 9 11 13 15 17 19
0 2 4 6 8 10 12 14 16 18 20 22 24 1 3 5 7 9 11 13 15 17 19 21
0 2 4 6 8 10 12 14 16 18 20 22 24 1 3 5 7 9 11 13 15 17 19 21 23
Resize function works properly
Number 6 present in Array
Number 30 not in Array
Number 23 removed from Array
Number 24 removed from Array
Number 0 removed from Array
Number not in Array
Array destroyed
```

The code provided to you, however, has lots of bugs in the implementations in `terrible_dynamic_size_array_unsorted.c` when you compile your code using the command:

```
gcc test_int_array.c terrible_dynamic_size_array_unsorted.c -w
```

Your job is to use `gdb` **ONLY** to debug the program and find all the bugs in `terrible_dynamic_size_array_unsorted.c`.

Note

- There's no bug in the `test_int_array.c` and `terrible....h`, so you are not allowed to modify these two files;
- You are also not allowed to use `printf` to debug. You should not add or modify any part of the `terrible...c` **unless** it's the bug you found.

What to Submit

At the end of the lab, you should submit your modified version of `terrible...c`. It doesn't need to be complete, but you should at least find out **2** bugs, and modify them, and comment them in the code.
Before **9/22 11:59PM**, you should submit a complete zip file, including the following items:

(1) Modified `terrible...c`: all bugs should have been modified right in the code, and pointed out using comments;
(2) A **PDF** lab report: for each bug you found, you should provide a screenshot of `gdb` and a brief description of how you used `gdb` commands to find the bug. Lastly, a screenshot of the successfully executed program without any errors or bugs. Any report in non-PDF format is not allowed.