- CPSC 275: Introduction to Computer Systems

CPSC 275: Introduction to Computer Systems

Fall 2025

- Syllabus
- Schedule
- Resources
- Upload
- Solution

Lab 7: Working with gdb

Objectives

The main goal of this laboratory is to:

1. Learn to use `gdb` to control and inspect program execution.
2. Practice debugging C and assembly programs using breakpoints and stepping commands.
3. Understand how to analyze and fix logical and run-time errors using `gdb`.

Introduction

The classic debugging technique is to simply add trace code to the program to print out values of variables as the program executes, using `printf` statement, for example. This approach requires a constant cycle of strategically adding trace code, recompiling the program, running the program, and analyzing the output of the trace code, removing the trace code after the bug is fixed, and repeating these steps for each new bug that is discovered. This is a highly time-consuming process.

Unfortunately, this approach may not even work at all in some situations. Suppose, for example, that your program crashes with a *segmentation fault*, that is, a memory access error. When this happens, the call to `printf` may not succeed because of the abnormal termination of the program. As a result, it is difficult to tell the approximate location of a bug.

A *debugger* is a software tool that allows you to control the execution flow of your program and access the state of the running program, including the content of the memory. The most commonly used debugger among Linux programmers is `gdb`. In this lab, you will explore some basic features of `gdb` and learn various debugging techniques which we will use throughout this term.

Part I: Getting Started with gdb

First, open a terminal and start `gdb`:

```
$ gdb
```

This will open a new `gdb` session with the following prompt:

```
(gdb)
```

Enter each of the following `gdb` commands and record and explain the output.

```
print/d 2025
p/d 2025
p 2025
p/t 2025
p/x 2025
p/d 0xf00d
p/t 0xf00d
p/d 0b1001
p/x 0b10101111
p/t 0b10101111 + 0xc1
p/t -10
p/t (char)(-10)
p/t (short)(-10)
quit
```

A detailed list of `gdb` commands can be found here.

Part II: Debugging C Programs

Download a C program, `prog1.c`, and compile it with the `-g` flag:

```
$ gcc -m32 -g -o prog1 prog1.c
```

This option adds debugging information to the executable for `gdb` to use. Now open your program with `gdb`:

```
$ gdb ./prog1
```

Run the program with:

```
(gdb) run
```

What happened? Now set a *breakpoint* at `main` and run:

```
(gdb) break main
Breakpoint 1, main () at prog1.c:6
(gdb) run
```

What happened now? By setting breakpoints, you may pause the execution of your program.

You can display the C source code around the current position using the `list` command:

```
(gdb) list
```

or around a specific line:

```
(gdb) list 14
```

Let's execute the program line by line using the `next` (or just `n` for short) command:

```
(gdb) next
7               int size = 5;
```

The `next` command executes the current line and displays the next line to execute. Enter the `next` command again:

```
(gdb) next
8               int arr[] = {10, 20, 30, 40, 50};
```

Let's print the value of `size`:

```
(gdb) print size
$1 = 5
```

Here, $*i*$ is an internal variable which keeps track of the values in the `gdb` session.

Run your program until it terminates:

```
(gdb) continue
Continuing.
sum = 150
[Inferior 1 (process 2488862) exited normally]
```

Let us now set another breakpoint at the function `sum`:

```
(gdb) break sum
Breakpoint 2 at 0x5555555551ef: file prog1.c, line 15.
```

and run the program:

```
(gdb) run
Starting program: /home3/pyoon/prog1
Breakpoint 1, main () at prog1.c:6
6       {
(gdb) continue
Breakpoint 2, sum (a=0x0, n=21845) at prog1.c:15
15      int sum(int a[], int n) {
```

Run this line:

```
(gdb) next
```

Enter `gdb` commands to print the values of `a`, `n`, `a[2]`, and `&a[2]`.

Now, enter the `next` command repeatedly and each time, print the values of `i` and `total`. Stop when you see:

```
(gdb) next
19              return total;
```

Enter the `next` command twice. Which line do you see? Explain.

Finally, enter the `continue` command to run the program until it terminates. Enter the command `quit` to terminate the current session:

```
(gdb) quit
```

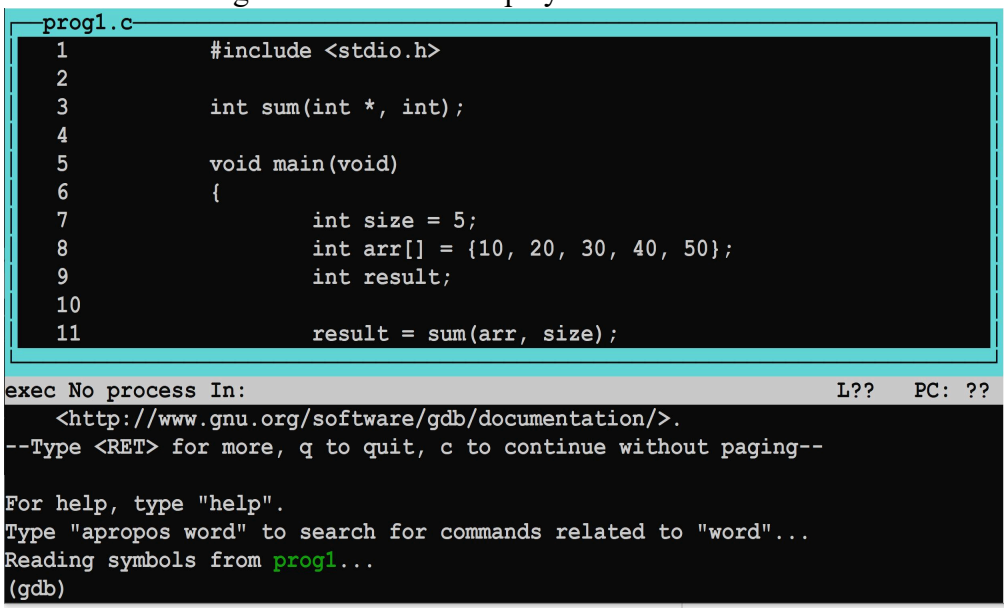
Part III: The TUI Interface

The `gdb` comes with a terminal interface that shows the source file, the assembly output, the program registers, and `gdb` commands in separate text windows.

Open `gdb` with the `-tui` flag:

```
$ gdb -tui prog1
```

Then the following terminal will be displayed:



Repeat Part II in TUI (Text User Interface) mode.

Part IV: Debugging IA-32 Assembly Programs

Download num.s, and compile it with the `-g` flag:

```
$ gcc -m32 -g -o num num.s
```

Now open your program with `gdb`:

```
$ gdb ./num
```

Set a breakpoint at `main`:

```
(gdb) break main
```

Run the program:

```
(gdb) run
```

Execute one instruction:

```
(gdb) stepi
```

Execute another instruction:

```
(gdb) stepi
```

Observe register changes:

```
(gdb) info registers
```

Print the value of `%eax` (Note that `$` is used to represent registers in `gdb`):

```
(gdb) print /d $eax
```

Execute one instruction:

```
(gdb) stepi
```

Print the value of `%eax`:

```
(gdb) print /d $eax
(gdb) print /x $eax
```

Disassemble the function `main`:

```
(gdb) disas main
```

Note that the `=>` on the left indicates the current instruction. Let's check the message with the label `.LC0` in memory (But where is the address of the string?):

```
(gdb) print (char *) addr_of_the_string
(gdb) x/s addr_of_the_string
```

Execute one instruction:

```
(gdb) stepi
```

Execute one instruction:

```
(gdb) stepi
```

Examine the stack:

```
(gdb) x/wd $esp
```

What's on top of the stack?

Execute one instruction:

```
(gdb) stepi
```

Examine the stack:

```
(gdb) x/wd $esp
```

What's on top of the stack?

Execute one instruction:

```
(gdb) stepi
```

Examine the stack:

```
(gdb) x/wx $esp
```

What's on top of the stack?

Execute one instruction:

```
(gdb) stepi
```

Examine the stack:

```
(gdb) x/wx $esp
```

What's on top of the stack?

Print the value of `%eip`:

```
(gdb) x/wx $eip
```

What address is in `%eip`?

Run until the end:

```
(gdb) continue
```

Stop `gdb`:

```
(gdb) quit
```

V. Exercises

The following programs either contain logical errors or terminate with run-time errors:

- prog2.c
- prog3.c
- prog4.c
- prog5.c

Compile and run each program. If you see any errors, recompile it with the `-g` flag and find and correct the bugs using `gdb`.

When completed, ask your instructor or TA to check your work.

- **Welcome: Sean**
  - LogOut