GDB TUTORIAL

What is GDB ?

GDB, part of GNU open source, is a debugger for programming languages like C, C++, Assembly, FORTRAN, etc.

Okay..! But why is debugger important..? 🤔

Have you ever received Segmentation fault and wondered where you went wrong? Or received unintended output and been in a position, unable to explain where the variables got modified ? Then debugger is exactly what you should be looking for.

Debuggers are software tools which help to identify bugs or defects which are causing unintended behaviour or incorrect results. They help you to inspect the state of a **running program,** track the flow of execution and analyze variables, data structures and memory contents. So, basically you are able to run a program with complete control over it.

For the sake of this tutorial we will be using the following program : **test.c**

```
#include <stdio.h> int add(int a , int b) { int res = a + b; return res; }
int increment(int a) { a = a++; return a; } int main() { int x, y; unsigned
int blah = 4294967291; int m = 10; printf("Enter two numbers for addition :
"); scanf("%d %d", &x, &y); int ans = add(x, y); printf("Sum of two numbers
is : %d\n", ans); printf("Value of blah is : %d\n", blah); int arr[5] = {1,2
,3,4,5}; int z; printf("Enter a number to increment: "); scanf("%d", &z); z
= increment(z); printf("Incremented number: %d\n", z); if(m = 11) {
printf("m is 11\n"); } else { printf("m is not 11\n"); } return 0; }
```

Some commands of GDB and their uses

1. GDB Initialisaion

```
gcc -g filename
```

- This will create an executable file like a.out and `-g` flag ensures that the, executable file so created is runnable by GDB

2. Loading the executable

```
gdb executable_file_name
```

- Executable file will be opened in GDB environment but **is not yet being executed.**

3. Running the executable

```
[r]un
```

- Square bracket on r implies either you can just type 'r' and press enter or type 'run' and press enter. Both will work. Square bracket in all subsequent instructions means the same.
- Oh no !😑 Program just ran and got completed. We had no control on it at all. You can do the following to gain control over the program.

4. Controlling the program

```
[b]reak argument
```

- This will set a breakpoint in the program i.e the program will run till the breakpoint without any pauses in between and will stop execution as soon as it reaches the defined breakpoint. And from that point onwards you can control the program instruction by instruction.
- Valid arguments :
  - function name : **Ex:** `b main` , `b add`
  - line number : **Ex:** `b 6` (Stop execution as soon as the control reached line number 6)
- Ideally we set the breakpoints before the run command. But you can also set new breakpoints even after the program execution begins.

- break statements assign an identifier to each breakpoint in the form of natural numbers. (1,2,3....)

```
[n]ext
```

- Execute the current instruction and move to the next line of instruction.

```
[s]tep
```

- Step into the current instruction and then move to the next line of instruction.
- Whats' the difference between `next` and `step` 🤔?
  - Lets say control is presently at a user-defined function call. Ex: add(3, 4)
  - Now if you use `next` command, it will get you the result of this function and move to the next instruction which is printf statement in the example code.
  - But if you use `step` command, control will move into the definition of add function and you can execute the function also line by line, then come back to place from where function was called and execute the next line of instruction.
  - If its not a function call then there is no difference.
- Inorder to go step by step over the body of any function you can also set a breakpoint at the function and use `next` statement. It will be exactly same as using `step` at any function call.

```
[c]ontinue
```

- Continue the execution of the program until it encounters the next breakpoint, watchpoint or the program terminates

---

```
finish
```

- Continue the execution of the program until the current function returns.
- Its not meaningful in outermost frame i.e main functtion. It can be used inside any other user defined function to execute until it returns

---

```
[p]rint argument
```

- It can be used to check the value of any variable or register in between the program execution.
- Arguments can be :
  - variable name. Ex: `print x`
  - register name. Ex: `print $rax` ($ symbol is must before the name of the register.)

```
set varname = value
```

- We can change the value of any predefined variable to the desired value during the program execution using the set command.
  - Ex: `set ans = 3`

```
[wat]ch varname
```

- This is a kindof conditional breakpoint
- It says pause the execution of the program as soon as the value of the variable `varname` changes.
- Ex: `watch ans` Pause the execution of the program if value stored in ans variable changes.
- It is advised that use watch on any variable only after the variable is initialised.

```
[d]elete argument
```

- Used to delete already set breakpoints.

- argument should be a number (id of the breakpoint)
- If no argument is provided it will delete all the breakpoints

---

```
[i]nfo argument
```

- It is used to get information about the given argument.
- Arguments can be :
  - `locals` : This will print the value of all the local variables in the form varname = value.
  - `breakpoints` : This will give info about all the breakpoints along with their ids.
  - `watch` : This will give you list of all the watchpoints set in the program.
  - `registers` : This will give you values stored in all the registers.
    - **You will be using this a lot in your assignments 😯😯**

```
backtrace or bt
```

- This is used to view the present call stack.
- Basically it will list all the function calls present in the program stack in the reverse order.

```
[k]ill
```

- This will kill the current program being executed that is it will terminate the program execution, but it will not exit from gdb.

```
[q]uit
```

- This will take you out of GDB.

```
disassemble argument
```

- This will generate assembly code for the given argument.
- Arguments:
  - function name Ex: `disassemble add`
  - address range
  - Passing no arguments will give you assembly code for the current function in which control exists.

⚠️ **You can use this to understand what's happening under the hood and it comes with lot of optimisations. Feel free to experiment but won't recommend using it for the assignment.**

✅✅ **Compile the program `test.c` provided at the very beginning of this document and try to debug it using gdb.**

▼ Hint

- There are 3 errors and none of them are compilation errors 🤭🤭
