Centre for Advanced Studies
Secure Software Design and Operating System Security
(MCySC-202)
Second Semester 2017-2018
Lab-1 (GDB Tutorial).
#Requirement of the exercise is a Linux OS and gcc compiler.

Objectives:

- Introduction
- Getting started with GDB (GNU Debugger)
- Basic Commands
- A sample run
- Exercise (choose any program which can cause more run time errors)

Introduction:

The debugger gdb is a powerful source-level debugger for all the languages that gcc compiles, and it is programmable itself! You can even write scripts for automating debugging and test-data collecting tasks in gdb non-interactively.

The purpose of a debugger such as gdb is to allow you to see what is going on “inside” another program while it executes—or what another program was doing at the moment it crashed. gdb can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Getting started with gdb:

➔You need to compile the code with the -g flag, among others. (i.e. $ gcc -g pgm1.c -o pgm1)

➔Then the executable binary generated can be debugged with gdb. Or ddd.(i.e. $ gdb ./pgm1)

➔For all languages compiled by the GCC, the GNU COMPILER COLLECTION, gdb can be used. e.g. C++ (using the g++ compiler), Java (using gcj), Fortran (gfortran), etc

Basic Commands:

Before moving on, let's have a look at command you will often use.

run

This starts the program. If you haven't set breakpoints the program will run until it terminates (normally or by crashing). Its shorthand is r.

backtrace

Shows your current location in the program as well as a stack trace showing how you got there. It is very useful when you want to trace the function calls up until a segfault. Its shorthand is bt.

break

This sets up a breakpoint. Breakpoints allow you to stop execution temporarily so that you can examine the values of variables, functions etc.. at that point. You may have several breakpoints in your progams and there are several ways to set one:

- break *function_name*

Sets a breakpoint at the beginning of the function.
Ex: break main
break sum

- break *filename:line_number*

Sets a breakpoint at a specified line of the given file.
Ex: break pgm1.c:10

continue

If you type continue or c after having stopped at breakpoint, execution continues until program terminates or another breakpoint is encountered.

info breakpoints

This lists the current breakpoint. You can delete that particular breakpoint by typing delete and the number of that breakpoint.

next

This executes the next command. Its shorthand is n.

step

This steps through the next command. The difference between next and step command is that if the next command contains a function call, than typing the next will execute it without stepping through it, while typing step will take you to the first line of the function. Its shorthand is s.

print

This prints out the value of the variable or expression. The shorthand is p.

Ex: print num

list

This shows you the code surrounding the current line. Its shorthand is l.

info locals

This shows the values of all the local variables with in a function.

quit

Quit gdb.

A sample run

Let's run a factorial program with segmentation fault in order to understand the functionalities of gdb.

```
#include<stdio.h>

void main()
{
        int n;
        printf("Enter the number\n");
         scanf( "%d" , &n );
        int res = fact( n );
        printf("The factorial is %d " , res );
}

int fact ( int num )
{
        return num * fact( num - 1 );
}
```

compile: gcc -g factorial.c -o factorial

debug: gdb ./factorial

```
(gdb) r
Starting program: /home/factorial
Enter the number
4

Program received signal SIGSEGV, Segmentation fault.

 0x000000000040060d in fact (
        num=<error reading variable: Cannot access memory at address 0x7fffff7feffc>) at factorial.c:11

(gdb) break main
Breakpoint 1 at 0x4005bc: file fact-seg.c, line 5.


(gdb) info breakpoints
Num Type           Disp      Enb  Address                 What
1     Breakpoint   Keep      y    0x00000000004005bc      in main at factorial.c:5

(gdb) r
Starting program: /home/shamanth/fact-seg

Breakpoint 1, main () at fact-seg.c:5

        printf("Enter the number\n");
(gdb) next
Enter the number
        scanf( "%d" , &n );
(gdb) n
4
            int res = fact( n );
```

```
(gdb) info locals
n = 4
res = 0

(gdb) print n

$ 1 = 4

(gdb) p res

$ 2 = 0

(gdb) step
fact (num=4) at fact-seg.c:12
                return num * fact( num - 1 );
(gdb) s
fact (num=3) at fact-seg.c:12
                return num * fact( num - 1 );
(gdb) s
fact (num=2) at fact-seg.c:12
                return num * fact( num - 1 );
(gdb) s
fact (num=1) at fact-seg.c:12
                return num * fact( num - 1 );
(gdb) s
fact (num=0) at fact-seg.c:12
                return num * fact( num - 1 );
(gdb) backtrace
#0 fact (num=0) at fact-seg.c:12
#1 0x000000000040061d in fact (num=1) at fact-seg.c:12
#2 0x000000000040061d in fact (num=2) at fact-seg.c:12
#3 0x000000000040061d in fact (num=3) at fact-seg.c:12
#4 0x000000000040061d in fact (num=4) at fact-seg.c:12
#5 0x00000000004005e9 in main () at fact-seg.c:7

(gdb) s
fact (num=-1) at fact-seg.c:12
                return num * fact( num - 1 );
(gdb) s
fact (num=-2) at fact-seg.c:12
                return num * fact( num - 1 );
(gdb) s
fact (num=-3) at fact-seg.c:12
                return num * fact( num - 1 );
(gdb) s
fact (num=-4) at fact-seg.c:12
                return num * fact( num - 1 );

(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
```

0x000000000040060d in fact (
num=<error reading variable: Cannot access memory at address 0x7fffff7feffc>) at factorili.c:11
(gdb) q
A debugging session is active.

Inferior 1 [process 14946] will be killed.

Quit anyway? (y or n) y

Now by using gdb you might have realized that segmentation fault had occurred mainly due to stack overflow now correct the code and try to debug it.