C-Refresher: Session 02
GNU Debugger

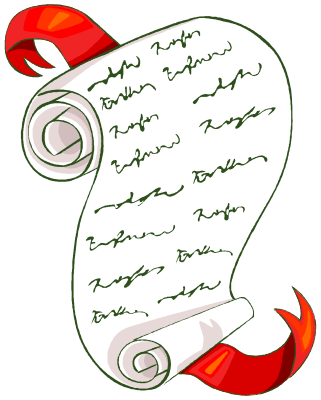
Arif Butt
Summer 2017

I am Thankful to my student Muhammad Zubair bcsf14m029@pucit.edu.pk for preparation of these slides in accordance with my video lectures at

http://www.arifbutt.me/category/c-behind-the-curtain/

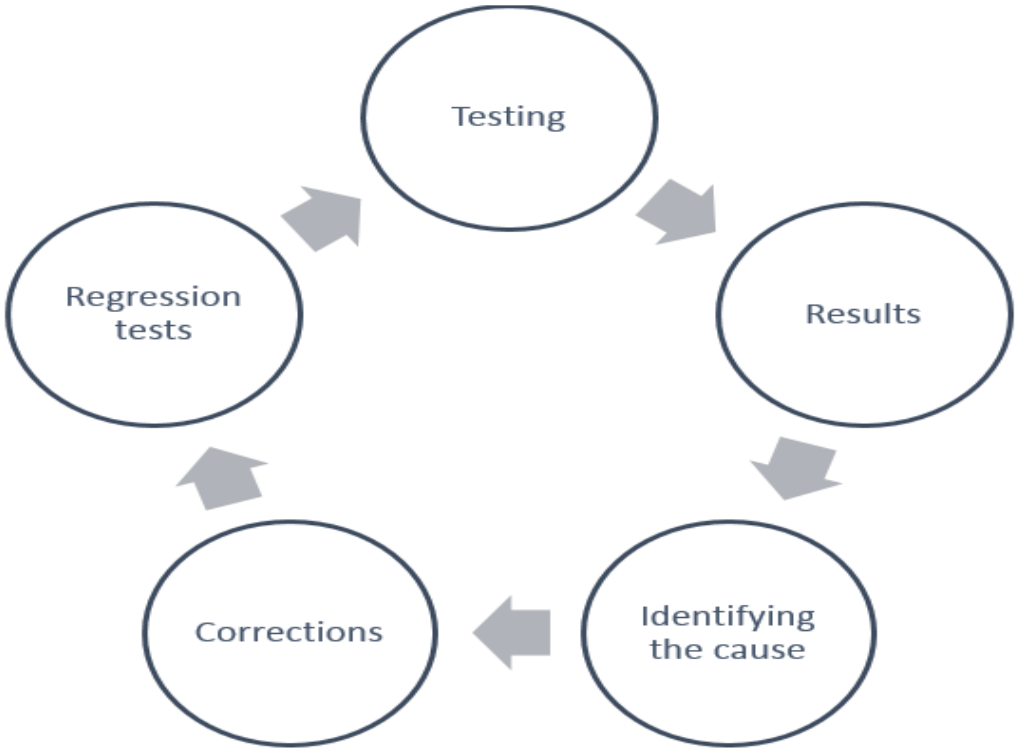
Today's Agenda

- Debugging
- gdb Debugger
- Working with gdb
- Finding Bugs in a Program
- Assembly Code of Program using gdb Debugger
- Working of Stack



Debugging

Debugging is a science or art of eliminating the bugs in a computer program.
There are a lot of debugging tools having both the command line and GUI interfaces



Debugging a cyclic process

Debugging(cont…)

❑Debugger:

Debugger is a computer program running another computer program in it. A debugger assists in the detection and correction of errors in a computer program.

❑Types of Debuggers:

`gdb` - the GNU debugger, Firefox JavaScript debugger, Microsoft visual studio debugger and many more.

gdb – the GNU debugger

- `gdb` allows 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 be used to debug programs written in C, C++, FORTRAN and Modula-2
- `gdb` allows you to run the program up to a certain point, then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line

gdb – the GNU debugger(cont…)

- **Four main things gdb can do**

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

Working with gdb

- For using `gdb` it must be installed first, if it is not installed, install it using command

```
$ sudo apt-get install libc6-dbg gdb valgrind
```

- Command to start `gdb`

```
$ gdb
```

- To avoid this additional info use `-q` option with `gdb` like

```
$ gdb -q
```

```
linux@ubuntu: gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 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".
(gdb)
```

Working with gdb(cont…)

- `gdb` is an interactive program, it waits for the commands from the user to execute
- To execute a shell command in `gdb` it must be proceeded with the bang sign(`!`)
- e.g.

```
(gdb) !clear
```

- `clear` is a shell command for clearing the screen, so it has been proceeded with the bang sign

Important gdb commands

Command	Description
`file [executable file name]`	To load a program executable file in `gdb`
`help`	Displays the classes of commands
`help [class name]`	List of commands in a specified class
`list`	Show the contents of the program loaded in `gdb`
`info inferiors`	Displays program(s) loaded in `gdb`
`add-inferior -exec [executable file name]`	To load more than one program in `gdb`
`inferior [program number]`	To switch to a specific program
`run [cmd line args]`	To run/execute the program with cmd line args if needed

Important gdb commands

Command	Description
`watch [variable name]`	Interrupts the execution of the program when the value of the variable changes
`break [line number]`	Apply break point at a specific line
`info break`	Displays the classes of commands
`continue` **or** `c`	To continue the program execution till the program end or the next breakpoint
`next` **or** `n`	To execute the next instruction
`backtrace` **or** `bt`	Displays the contents of the program stack
`finish`	To execute till the end of current function and return to the previous frame in stack

Working with gdb(cont…)

```
/*prog1.c…we will be using this example program for
understanding gdb commands*/
1. #include<stdio.h>
2. int main(){
3.     int n;
4.     printf("Enter a number: ");
5.     scanf("%d",&n);
6.     for(int i=0 ; i<n ; i++){
7.         printf("Learning Linux is fun!\n");
8.     }
9.     return 0;
10. }
```

Working with gdb(cont…)

- For a program to be loaded in `gdb` it must be compiled using `-g` or `-ggdb` option
- e.g.

```
$gcc -g prog1.c -o prog1 or $gcc -ggdb prog1.c -o prog1
```

❑Now the program can be loaded in `gdb` in three ways:

1. While starting `gdb` give the program executable filename as argument, like

```
$gdb prog1
```

2. Using `file` command of `gdb` after `gdb` has been started

```
(gdb) file prog1
```

Working with gdb(cont…)

3. Using `attach` command and giving `PID` of some running process as argument

- **Syntax**

  `(gdb) attach [PID]`

- Let's suppose we run a program `top`

  `$ top` //displays Linux processes

- We can get the `PID` of running processes using command

  `$ ps –au`

- Now the process can be loaded in `gdb` using command

  `(gdb) attach [PID]`

Working with gdb(cont…)

```
linux@ubuntu: ps -au
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       4201  0.0  0.0  23008   296 tty1     Ss+  Mar17   0:00 /sbin/agetty -
root       4220  0.6  2.9 451160 29392 tty7     Ss+  Mar17  12:59 /usr/lib/xorg/
zubair     5915  0.0  0.3  29728  3396 pts/22   Ss   Mar17   0:01 bash
zubair     6110  0.0  0.2  29544  2148 pts/17   Ss   Mar17   0:00 bash
zubair    13684  0.0  0.2  29504  2404 pts/4    Ss+  Mar17   0:00 bash
zubair    26198  0.2  0.3  48996  3764 pts/17   S+   10:20   0:00 top
zubair    26208  0.0  0.3  44432  3196 pts/22   R+   10:22   0:00 ps -au
linux@ubuntu:
```

- As we can see here that `top` command is running and its `PID` is `26198`
- So to load it in `gdb` we have to use the command
  `(gdb) attach 26198`

Working with gdb(cont…)

❑inferior

- `gdb` lets you load more than one programs in a single session and switch focus between them
- `gdb` does this with the object `inferior` like `inferior 1, inferior 2, inferior 3 …`
- Command for loading a process after the first process has been loaded is

```
(gdb) add-inferior –exec prog2        /*where
```

`prog2` is the name of executable file for `prog2.c`*/

Working with gdb(cont…)

- Command used to show loaded programs is

```
(gdb) info inferiors
```

- Command to switch focus from one program to another is

```
(gdb) inferior [inferior number]
```

Working with gdb(cont…)

```
linux@ubuntu: gdb -q prog1
Reading symbols from prog1...done.
(gdb) info inferiors
  Num  Description       Executable
* 1    <null>            /home/zubair/Documents/slides/gdb/prog1
(gdb) add-inferior -exec prog2
Added inferior 2
Reading symbols from prog2...done.
(gdb) info inferiors
  Num  Description       Executable
* 1    <null>            /home/zubair/Documents/slides/gdb/prog1
  2    <null>            /home/zubair/Documents/slides/gdb/prog2
(gdb) inferior 2
[Switching to inferior 2 [<null>] (/home/zubair/Documents/slides/gdb/prog2)]
(gdb) info inferiors
  Num  Description       Executable
  1    <null>            /home/zubair/Documents/slides/gdb/prog1
* 2    <null>            /home/zubair/Documents/slides/gdb/prog2
(gdb)
```

An example showing working of inferior command

Working with gdb(cont…)

❑run

- After a program has been loaded in `gdb`, it can be executed using `run` command
- **Syntax**

```
(gdb) run [cmd line arguments]
```

- **Note:** If there are more than one programs loaded in `gdb` then only one program can be executed at a time and the program having focus on it will be executed by `run` command

Working with gdb(cont…)

❑list

- `list` command is used to display the contents of the program loaded in `gdb` and currently having focus on it
- e.g.

```
linux@ubuntu: gdb -q prog1
Reading symbols from prog1...done.
(gdb) list
1       #include<stdio.h>
2
3       int main(){
4               int n;
5               printf("Enter a number: ");
6               scanf("%d",&n);
7               for(int i=0 ; i<n ; i++){
8                       printf("Learning Linux is fun!\n");
9               }
10
(gdb)
```

Working with gdb(cont…)

❑help

- `help` command in `gdb` is used to display the list of classes of commands

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
```

Working with gdb(cont…)

- Syntax to display the list of commands in a class is

  `(gdb) help [class name]`

  - e.g. `(gdb) help data`//display all commands in data class
- Syntax to display the list of all commands

  `help all`
- Syntax to display the documentation of a specific command is

  `(gdb) help [command name]`

  - e.g. `(gdb) help list` /* will display a full documentation of `list` command */

Working with gdb(cont…)

❑print

- `print` command is used to print the value of an expression or variable passed as argument to it
- `print` command is generally used to print the values of variables while debugging the program
- **Syntax**

```
(gdb) print [expression/variable name]
```

- e.g. standing at line 7 in `prog1.c` to check the value of `n` and `i` we will type

```
(gdb) print n    //result will be the value entered
(gdb) print i    //result will be 0 for the first time
```

Working with gdb(cont…)

- To print the value in hex format use `/x` option with the `print` command like

```
(gdb) print /x n        /*will print the value of n in hex format*/
```

- To print the value in octal format use `/o` option with the `print` command like

```
(gdb) print /o n        /*will print the value of n in octal format*/
```

- To print the value in binary format use `/t` option with the `print` command like

```
(gdb) print /t n        /*will print the value of n in binary format*/
```

Working with gdb(cont…)

- To print the value of some register in hex use `/x` option with print command and also use `$` sign with the name of the register
- e.g. **`rip`** is a register that holds the address of the next instruction to be executed, to display its value we can write

```
(gdb) print /x $rip
```

Working with gdb(cont…)

❑whatis

- `whatis` is a `gdb` command used to display the datatype of some variable
- e.g.

```
(gdb) whatis n   //will display the following result
type = int
```

❑set

- `set` is a `gdb` command used to temporarily set the value of some variable for debugging purposes
- e.g. `(gdb) set variable i=5` or `(gdb) set [i=5]` /*will set the value of `i=5` at that point*/

Working with gdb(cont…)

❑Watchpoint:

- A watchpoint pauses execution of a program whenever the value of a certain expression/variable changes
- `watch` command is used to apply watchpoints
- **Syntax**

```
(gdb) watch [expression]
```

- e.g. in program `prog1.c`

```
(gdb) watch i  /*will apply watchpoint on variable i
```

and will interrupt whenever the value of `i` changes*/

Working with gdb(cont…)

❑Breakpoint:

- Breakpoint can be applied at any line or function by giving the line number or the function name as argument
- When we apply a breakpoint, it pauses the execution of the program when the program reaches that point
- `break` command of `gdb` can be used for applying breakpoints in a program
- **Syntax**

```
(gdb) break [line number/function name] or
(gdb) break prog1.c:[line number/function name]
```

- The second one should be used if there are more than one programs loaded in `gdb`

Working with gdb(cont…)

- Breakpoints may be more than one in a program
- To display the list of all breakpoints in a program use command

```
(gdb) info break
```

- To disable a breakpoint use command

```
(gdb) disable [break number]
```

- And to enable a breakpoint

```
(gdb) enable [break number]
```

- To delete a breakpoint

```
(gdb) delete [break number]
```

Working with gdb(cont…)

```
(gdb) info break
No breakpoints or watchpoints.
(gdb) break 6
Breakpoint 1 at 0x40066c: file prog1.c, line 6.
(gdb) break main
Breakpoint 2 at 0x40064e: file prog1.c, line 3.
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040066c in main at prog1.c:6
2       breakpoint     keep y   0x000000000040064e in main at prog1.c:3
(gdb) disable 1
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep n   0x000000000040066c in main at prog1.c:6
2       breakpoint     keep y   0x000000000040064e in main at prog1.c:3
(gdb) delete 2
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep n   0x000000000040066c in main at prog1.c:6
(gdb) enable 1
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040066c in main at prog1.c:6
(gdb)
```

Working with gdb(cont…)

❑continue

- `continue` is a `gdb` command that is used to continue the execution of program till the end or till some breakpoint
- **Syntax**

```
(gdb) continue or (gdb) c
```

❑next

- `next` is also a `gdb` command used to execute the very next program instruction/line
- **Syntax**

```
(gdb) next or (gdb) n
```

Working with gdb(cont…)

- **Note:**
  - In `gdb`, simply pressing `ENTER` will execute the command that was last executed
  - e.g.
  - we execute the command

    ```
    (gdb) next
    ```

  - After executing this command, no need to type `next` again to execute the `next` command, rather simply press `ENTER` and the `next` command will be executed again

Finding Bugs in a Program

- `gdb` can be used to find bugs in a program
- **Breakpoints** are the main key in finding bugs in a program

❑**Procedure -1:**

- Use `(gdb) next` command to execute each statement of the program
- Print the values of variables using the `(gdb) print` command
- Observe values of variables and get to the error and remove that error

Finding Bugs in a Program(cont…)

❑Procedure -2:

- Apply breakpoints at different points/lines in a program
- Run the program using `(gdb) run` statement, the program will pause its execution at first breakpoint
- At that breakpoint, print the values of variables using `(gdb) print` statement
- Use `(gdb) continue` statement to reach the next breakpoint
- Again print the values of variables
- Observe the values, get to the error(s) and remove it

Assembly Code of Program

- `gdb` can be used to see the assembly of a program
- After the program has been loaded in `gdb` its assembly can be seen using command

```
(gdb) disassemble [function name]
```

- To see assembly of each line separately use `/m` with `disassemble` like

```
(gdb) disassemble /m [function name]
```

- If `disassemble` command is used during the execution of program at some breakpoint then the line having arrow at its beginning indicates that this line is under execution

Assembly Code of Program(cont…)

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000400646 <+0>:     push   %rbp
   0x0000000000400647 <+1>:     mov    %rsp,%rbp
   0x000000000040064a <+4>:     sub    $0x10,%rsp
   0x000000000040064e <+8>:     mov    %fs:0x28,%rax
   0x0000000000400657 <+17>:    mov    %rax,-0x8(%rbp)
   0x000000000040065b <+21>:    xor    %eax,%eax
   0x000000000040065d <+23>:    mov    $0x400744,%edi
   0x0000000000400662 <+28>:    mov    $0x0,%eax
   0x0000000000400667 <+33>:    callq  0x400510 <printf@plt>
=> 0x000000000040066c <+38>:    lea    -0x10(%rbp),%rax
   0x0000000000400670 <+42>:    mov    %rax,%rsi
   0x0000000000400673 <+45>:    mov    $0x400755,%edi
```

Snippet showing assembly using `disassemble` command

- The arrow sign at 6th line indicates that this line is currently under execution

Assembly Code of Program(cont…)

```
(gdb) disassemble /m main
Dump of assembler code for function main:
3       int main(){
   0x0000000000400646 <+0>:     push   %rbp
   0x0000000000400647 <+1>:     mov    %rsp,%rbp
   0x000000000040064a <+4>:     sub    $0x10,%rsp
   0x000000000040064e <+8>:     mov    %fs:0x28,%rax
   0x0000000000400657 <+17>:    mov    %rax,-0x8(%rbp)
   0x000000000040065b <+21>:    xor    %eax,%eax

4               int n;
5               printf("Enter a number: ");
   0x000000000040065d <+23>:    mov    $0x400744,%edi
   0x0000000000400662 <+28>:    mov    $0x0,%eax
   0x0000000000400667 <+33>:    callq  0x400510 <printf@plt>

6               scanf("%d",&n);
=> 0x000000000040066c <+38>:    lea    -0x10(%rbp),%rax
   0x0000000000400670 <+42>:    mov    %rax,%rsi
   0x0000000000400673 <+45>:    mov    $0x400755,%edi
   0x0000000000400678 <+50>:    mov    $0x0,%eax
```

Snippet showing result of `disassemble` with `/m`

Registers

- gdb also lets us know the values of different registers during the execution of the program
- Command used for this is

```
(gdb) info registers /*will display the values of different registers*/
```

❑Registers Details:

- For 64-bit architecture, there are 16 64-bit general purpose registers
- For 32-bit architecture, there are 8 32-bit general purpose registers

Registers(cont…)

- Then there is **IP** (Instruction Pointer) register which contains the address of the next instruction to be executed
- Below IP register, there is **eflags** register which contains bits of different flags like carry flag, sign flag, parity flag, zero flag
- Below that there are six segment registers namely **cs, ss, ds, es, fs, gs**
- Command to see all registers

```
(gdb) info all-registers
```

- This will show initial registers along with eight `80-bit` registers from **st0** to **st7** and then there are sixteen `256-bit` registers from **ymm0** to **ymm15**

Registers(cont…)

```
(gdb) info registers
rax            0x400646 4195910
rbx            0x0      0
rcx            0x0      0
rdx            0x7fffffffdf48   140737488346952
rsi            0x7fffffffdf38   140737488346936
rdi            0x1      1
rbp            0x7fffffffde50   0x7fffffffde50
rsp            0x7fffffffde40   0x7fffffffde40
r8             0x400730 4196144
r9             0x7ffff7de78e0   140737351940320
r10            0x846    2118
r11            0x7ffff7a2e740   140737348036416
r12            0x400550 4195664
r13            0x7fffffffdf30   140737488346928
r14            0x0      0
r15            0x0      0
rip            0x40064e 0x40064e <main+8>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

Registers(cont…)

❑Instruction Pointer(IP) Register

- IP points to the address of the next instruction to be executed

❑Function Calling

- When a function is called, the value of IP is pushed into a stack and the address of first instruction of function is stored in IP
- When the function terminates, the previously stored value in stack is popped out and assigned to IP register

❑rbp & rsp

- rbp points to the bottom of the current stack frame
- rsp points to the top of the stack, i.e. to the last occupied address by stack

Working of Stack

- When a program starts its execution, the `main()` function is pushed into a stack
- Whenever a function is called by the main, the called function is also pushed into the stack over the `main()` function
- And if this called function calls some other function, that called function is also pushed into that stack and so on
- When a called function has finished its execution, it is popped out from the stack

Working of Stack(cont…)

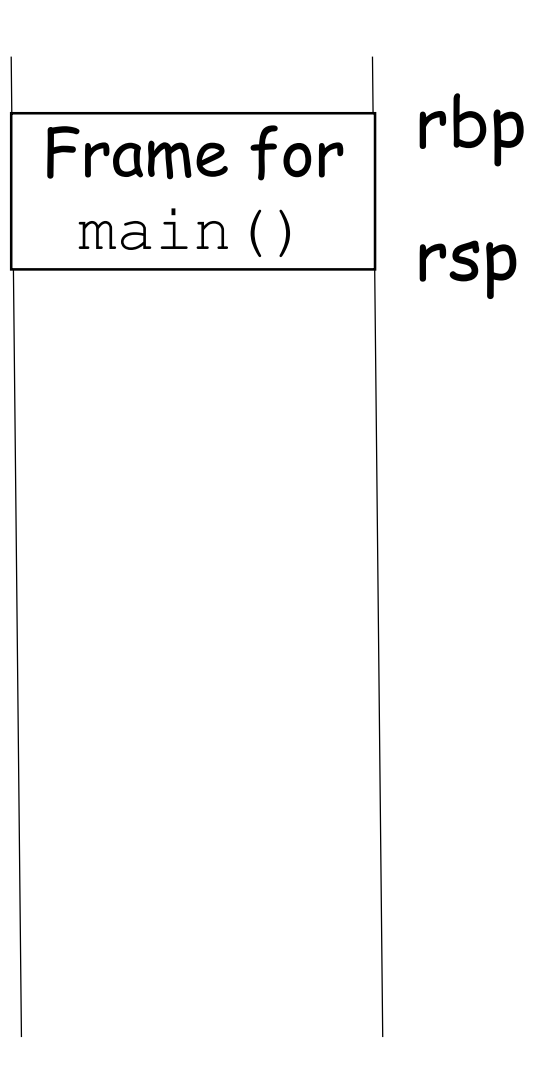
- rbp points to the address in the stack where last function has been pushed, whenever a new function is pushed into the stack rbp starts pointing to the start of that function
- And when a function is popped out from the stack, rbp starts pointing to the start of the function below it
- rsp points to the address in the stack where last byte of newly pushed function resides and whenever a new function is pushed into the stack, rsp starts pointing to its last byte
- And when a function is popped out of the stack, rsp starts pointing to the address of the last byte of the function below that popped function

Working of Stack(cont…)

```
/*stackDemo.c*/
1. #include<stdio.h>
2. int f1();
3. int f2();
4. int main(){
5.    f1();
8.    printf("DONE!\n");
9.    return 0;
10.}
11.int f1(){
12. f2();
13. return 1;}
14.int f2(){
15. return 2;}
```

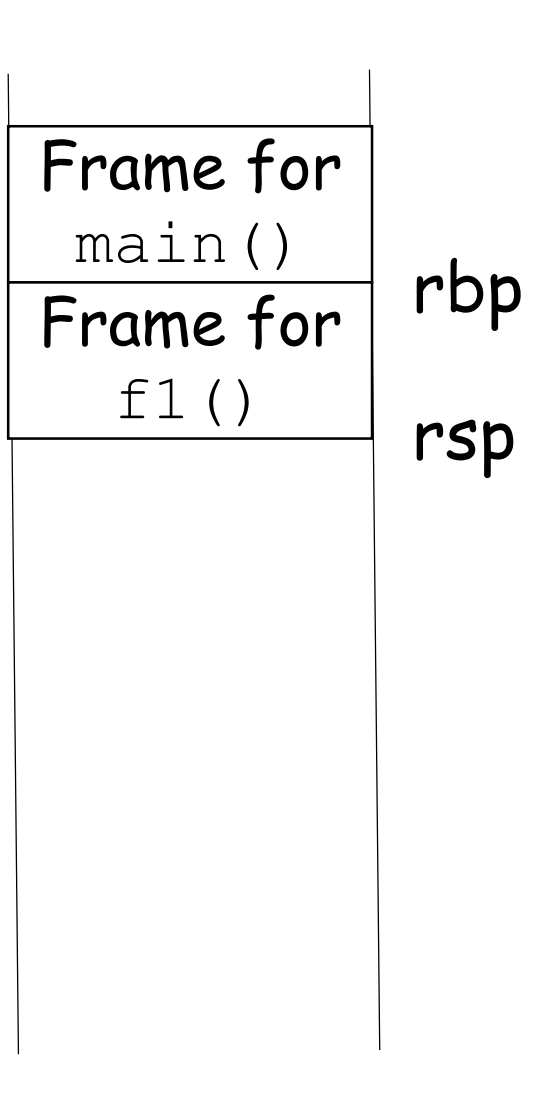
Growing of Stack

- At the start of the program the stack only contains `main()` function
- `rbp` and `rsp` are pointing to the start and end of `main()` function in the stack respectively



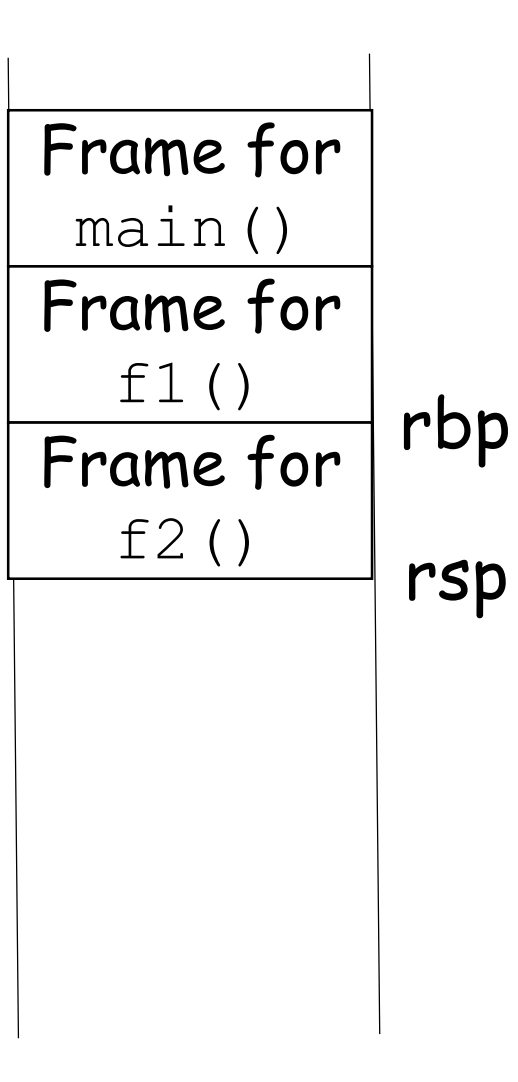
Growing of Stack(cont…)

- At line# 5, when function `f1()` is called, `f1()` is pushed into the stack over `main()`
- `rbp` and `rsp` now start pointing to the start and end of function `f1()` in the stack respectively



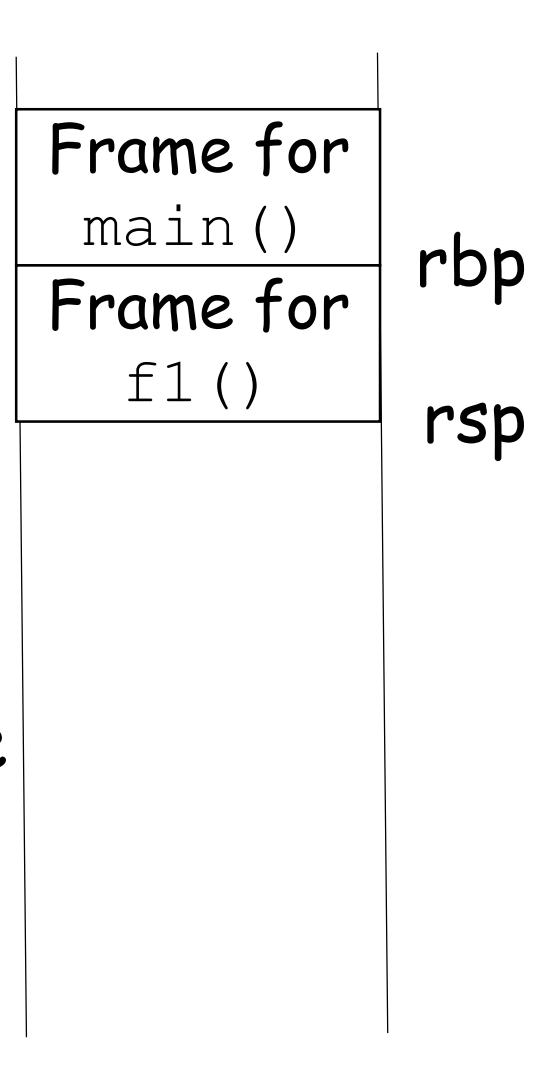
Growing of Stack(cont…)

- At line# 12, when function `f2()` is called, `f2()` is pushed into the stack over `f1()`
- `rbp` and `rsp` now start pointing to the start and end of function `f2()` in the stack respectively



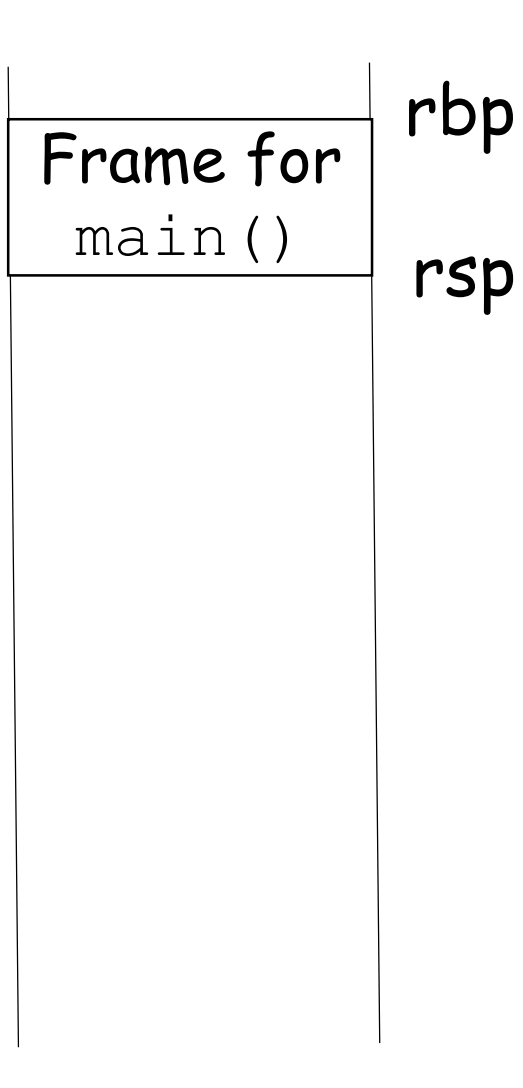
Growing of Stack(cont…)

- At line# 15, when return statement of `f2()` function is executed, it is popped out from the stack and control goes to line# 13 of `f1()`
- `rbp` and `rsp` now start pointing to the start and end of function `f1()` in the stack respectively



Growing of Stack(cont…)

- At line# 13, when return statement of `f1()` function is executed, it is popped out from the stack and control goes to line# 8 of `main()`
- `rbp` and `rsp` now start pointing to the start and end of function `main()` in the stack respectively



Growing of Stack(cont…)

- And finally at the end, when return statement of `main()` is executed at line# 9, it is also popped out from the stack and stack becomes **empty**

Stack commands

❑backtrace

- `backtrace` is used to print backtrace of all stack frames, i.e. to display all the contents of stack
- **Syntax**

  `(gdb) backtrace` or `(gdb) bt`

- To use `backtrace`, we can apply breakpoints at different points in the program and see the contents of stack at those break points using `backtrace`

Use of backtrace

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040052a in main at stackDemo.c:5
2       breakpoint     keep y   0x0000000000400549 in f1 at stackDemo.c:10
3       breakpoint     keep y   0x000000000040055e in f2 at stackDemo.c:13
(gdb) run
Starting program: /home/zubair/Documents/slides/gdb/stk

Breakpoint 1, main () at stackDemo.c:5
5               f1();
(gdb) bt
#0  main () at stackDemo.c:5
(gdb) c
Continuing.

Breakpoint 2, f1 () at stackDemo.c:10
10              f2();
(gdb) bt
#0  f1 () at stackDemo.c:10
#1  0x0000000000400534 in main () at stackDemo.c:5
(gdb) c
Continuing.

Breakpoint 3, f2 () at stackDemo.c:13
13              return 2;}
(gdb) bt
#0  f2 () at stackDemo.c:13
#1  0x0000000000400553 in f1 () at stackDemo.c:10
#2  0x0000000000400534 in main () at stackDemo.c:5
(gdb)
```

Stack commands(cont…)

❑finish

- `finish` command is used to return to the previous frame
- `finish` executes the current function, returns its value and stops over there
- **Syntax**

```
(gdb) finish
```

Use of finish

```
(gdb) bt
#0  f2 () at stackDemo.c:13
#1  0x0000000000400553 in f1 () at stackDemo.c:10
#2  0x0000000000400534 in main () at stackDemo.c:5
(gdb) finish
Run till exit from #0  f2 () at stackDemo.c:13
f1 () at stackDemo.c:11
11              return 1;}
Value returned is $1 = 2
(gdb) bt
#0  f1 () at stackDemo.c:11
#1  0x0000000000400534 in main () at stackDemo.c:5
(gdb) finish
Run till exit from #0  f1 () at stackDemo.c:11
main () at stackDemo.c:6
6               printf("DONE!\n");
Value returned is $2 = 1
(gdb) bt
#0  main () at stackDemo.c:6
(gdb) c
Continuing.
DONE!
[Inferior 1 (process 47073) exited normally]
(gdb)
```

SUMMARY