ATTACK DEFENSE

by PentesterAcademy

Name	Stopping and Continuing
URL	https://www.attackdefense.com/challengedetails?cid=2166
Type	Reverse Engineering : GDB Basics

Important Note: This document illustrates all the important steps required to complete this lab. This is by no means a comprehensive step-by-step solution for this exercise. This is only provided as a reference to various commands needed to complete this exercise and for your further research on this topic. Also, note that the IP addresses and domain names might be different in your lab.

Objective: Learn how to stop/run processes under GDB and check out different commands/options/methods.

Solution:

Settings Breakpoint

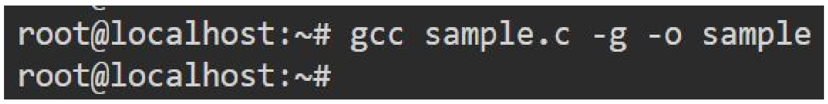
A breakpoint is to stop the execution when a certain point in the program is reached. One can also add conditions to it.

Generally, there are different types of breakpoints available, like:

- Breakpoints
- Watchpoints
- Catchpoints
- Tracepoints

Step 1: Compile the C file with GCC while enabling the debug symbols.

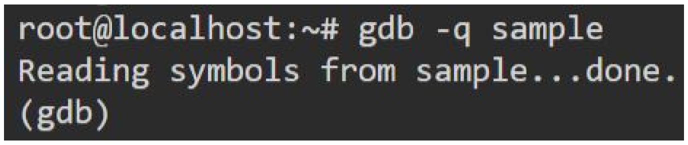
Command: gcc sample.c -g -o sample



```
root@localhost:~# gcc sample.c -g -o sample
root@localhost:~#
```

Step 2: Start GDB with the sample binary in quiet mode.

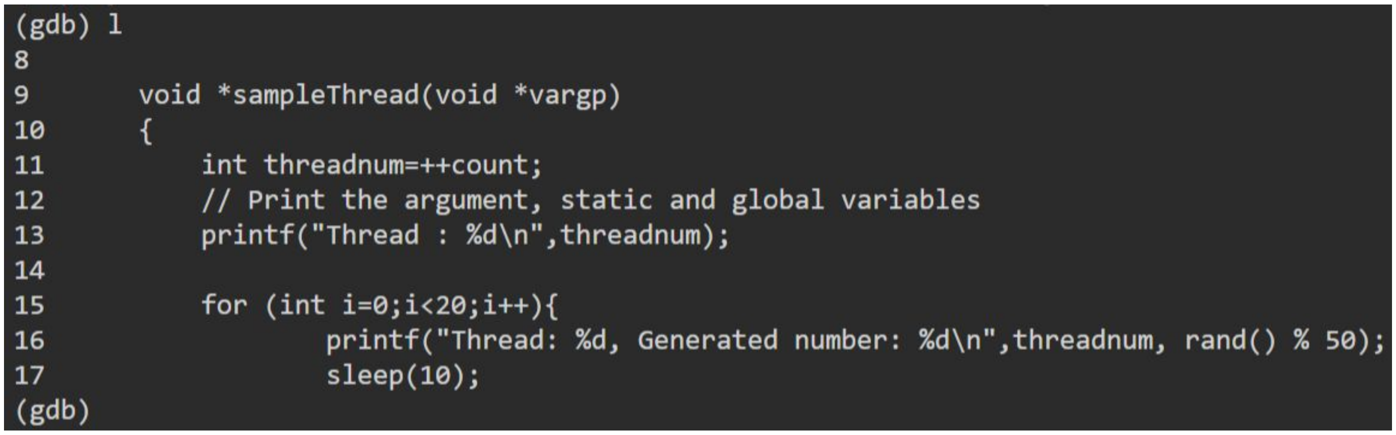
Command: gdb -q sample



```
root@localhost:~# gdb -q sample
Reading symbols from sample...done.
(gdb)
```

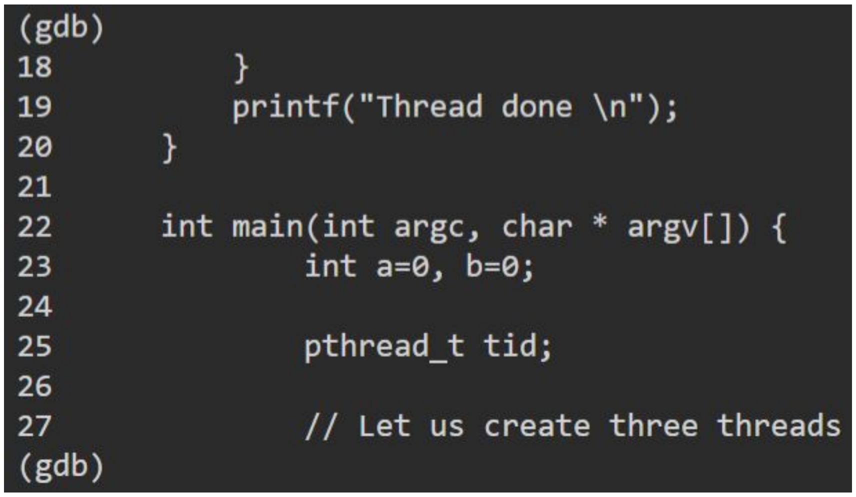
Step 3: Check the source code

Command: l   (small case L)

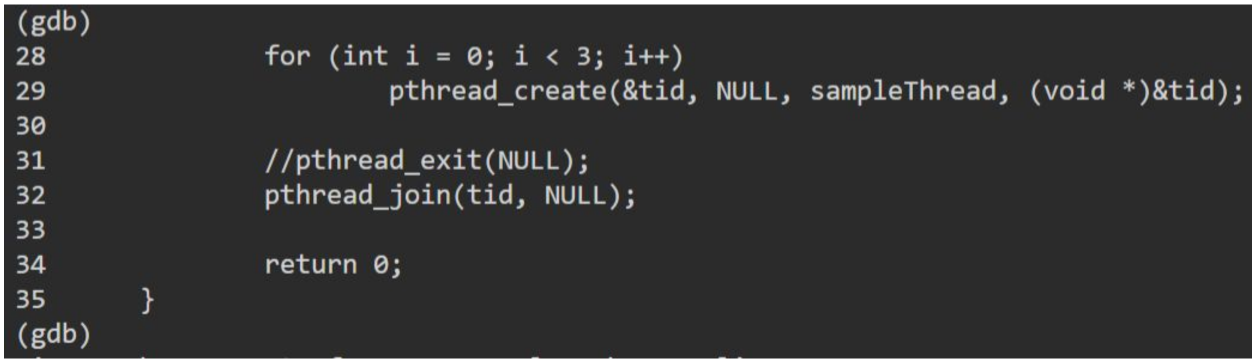


```
(gdb) l
8
9       void *sampleThread(void *vargp)
10      {
11          int threadnum=++count;
12          // Print the argument, static and global variables
13          printf("Thread : %d\n",threadnum);
14
15          for (int i=0;i<20;i++){
16                  printf("Thread: %d, Generated number: %d\n",threadnum, rand() % 50);
17                  sleep(10);
(gdb)
```

Pressing “enter” fires the last used command i.e. l (small case L) in this case.



```
(gdb)
18              }
19              printf("Thread done \n");
20      }
21
22      int main(int argc, char * argv[]) {
23              int a=0, b=0;
24
25              pthread_t tid;
26
27              // Let us create three threads
(gdb)
```

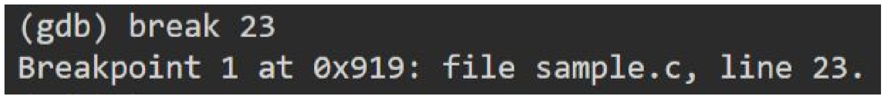


```
(gdb)
28              for (int i = 0; i < 3; i++)
29                      pthread_create(&tid, NULL, sampleThread, (void *)&tid);
30
31              //pthread_exit(NULL);
32              pthread_join(tid, NULL);
33
34              return 0;
35      }
(gdb)
```

Step 4: The Breakpoints can be added using different commands.

One can use the source line number (one visible in the source code listing above) can be used. Add a breakpoint at line 23

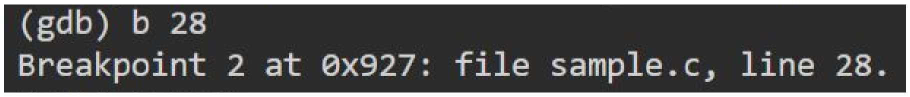
Command: break 23



```
(gdb) break 23
Breakpoint 1 at 0x919: file sample.c, line 23.
```

Similarly, the short-form of ‘break’ is b. Add a breakpoint at line 28

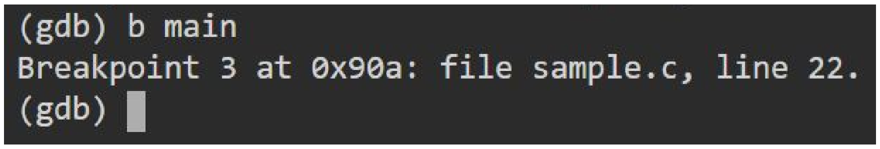
Command: b 28



```
(gdb) b 28
Breakpoint 2 at 0x927: file sample.c, line 28.
```

The function name can also be used to define the location of the breakpoint.

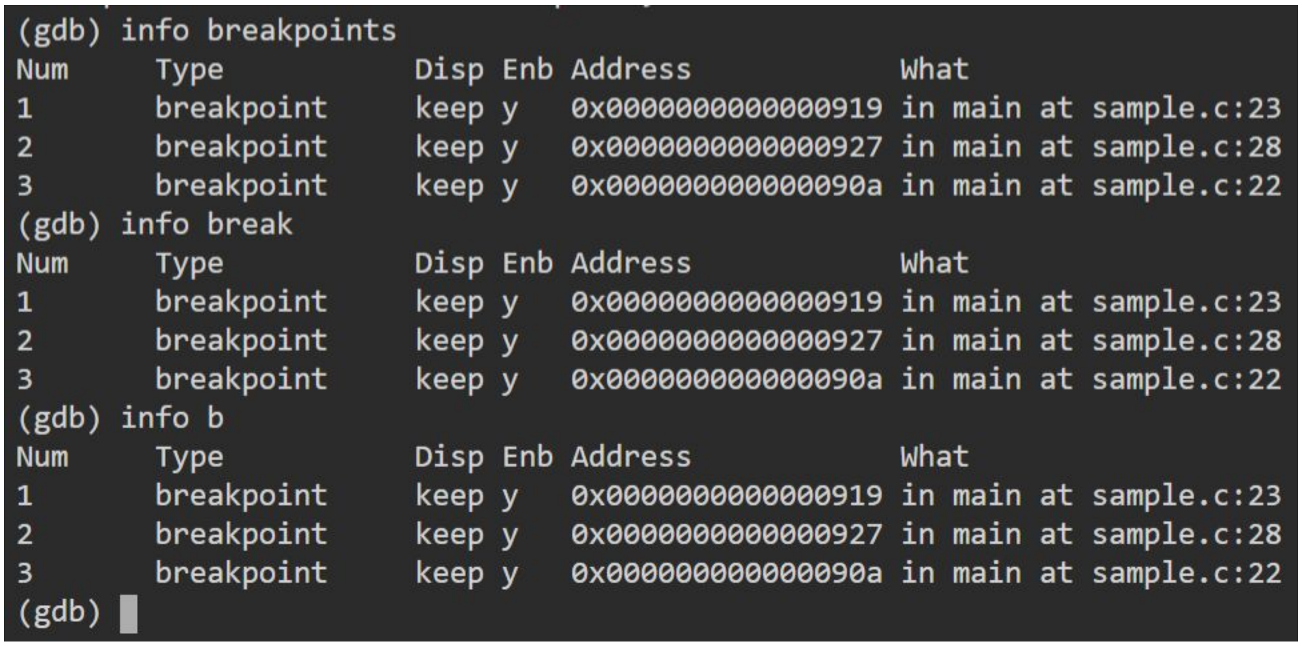
Command: b main



```
(gdb) b main
Breakpoint 3 at 0x90a: file sample.c, line 22.
(gdb) 
```

Step 5: List the added breakpoints Any of the following commands can be used to view breakpoints.

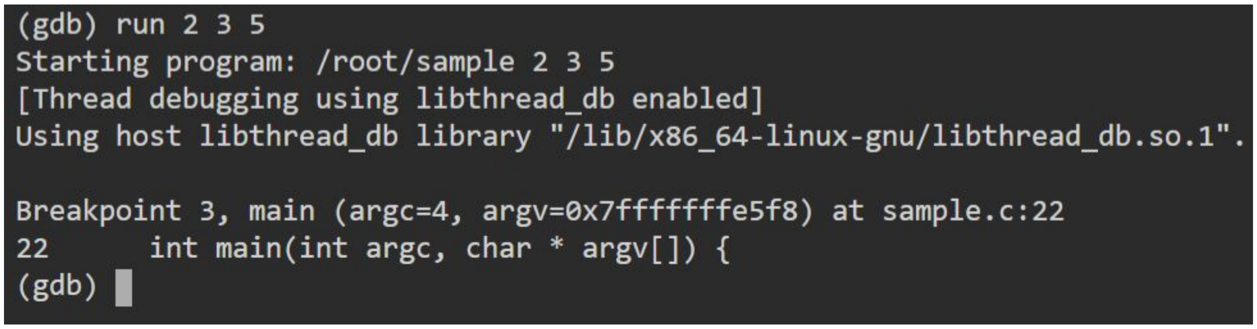
Commands:
info breakpoints
info break
info b



```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000000919 in main at sample.c:23
2       breakpoint     keep y   0x0000000000000927 in main at sample.c:28
3       breakpoint     keep y   0x000000000000090a in main at sample.c:22
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000000919 in main at sample.c:23
2       breakpoint     keep y   0x0000000000000927 in main at sample.c:28
3       breakpoint     keep y   0x000000000000090a in main at sample.c:22
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000000919 in main at sample.c:23
2       breakpoint     keep y   0x0000000000000927 in main at sample.c:28
3       breakpoint     keep y   0x000000000000090a in main at sample.c:22
(gdb) 
```

Step 6: Run the program. This program doesn't need any arguments so the arguments passed here won't have any impact.

Command: run 2 3 5

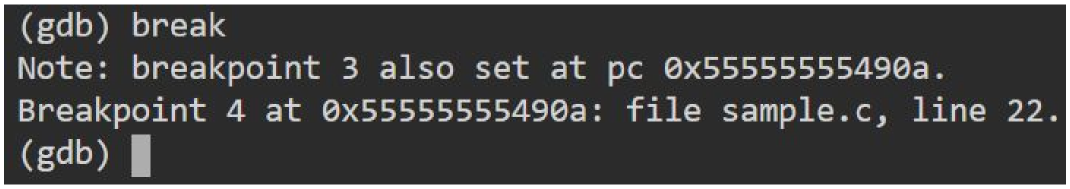


```
(gdb) run 2 3 5
Starting program: /root/sample 2 3 5
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 3, main (argc=4, argv=0x7fffffffe5f8) at sample.c:22
22      int main(int argc, char * argv[]) {
(gdb) 
```

The execution stopped at the first breakpoint 3 (it is the breakpoint added on 3rd time).

Step 7: Set a breakpoint on the next line.

Command: break

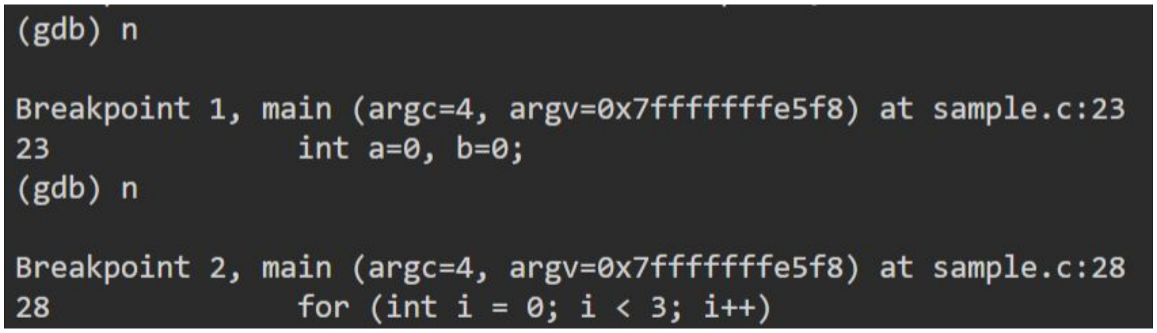


```
(gdb) break
Note: breakpoint 3 also set at pc 0x55555555490a.
Breakpoint 4 at 0x55555555490a: file sample.c, line 22.
(gdb) 
```

The breakpoint 4 is set on line number 22.

Step 8: Continue the execution to the next line of the source without stepping into the function.

Command: n



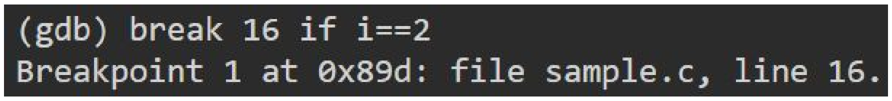
```
(gdb) n

Breakpoint 1, main (argc=4, argv=0x7fffffffe5f8) at sample.c:23
23              int a=0, b=0;
(gdb) n

Breakpoint 2, main (argc=4, argv=0x7fffffffe5f8) at sample.c:28
28              for (int i = 0; i < 3; i++)
```

The program execution stopped on breakpoint 1 and then breakpoint 2 on sending the ‘n’ or ‘next’ command.

__**Conditional Breakpoint**__

Step 9: Breakpoint can also be set to be triggered by a specific condition. Set a breakpoint on line 16 which should only be hit if the value of variable i (used in the loop) is equals to 2.

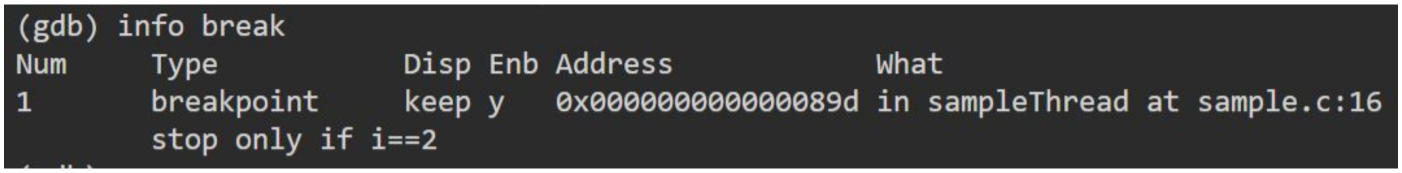
Command: break 16 if i==2



```
(gdb) break 16 if i==2
Breakpoint 1 at 0x89d: file sample.c, line 16.
```

Step 10: Check the set breakpoints

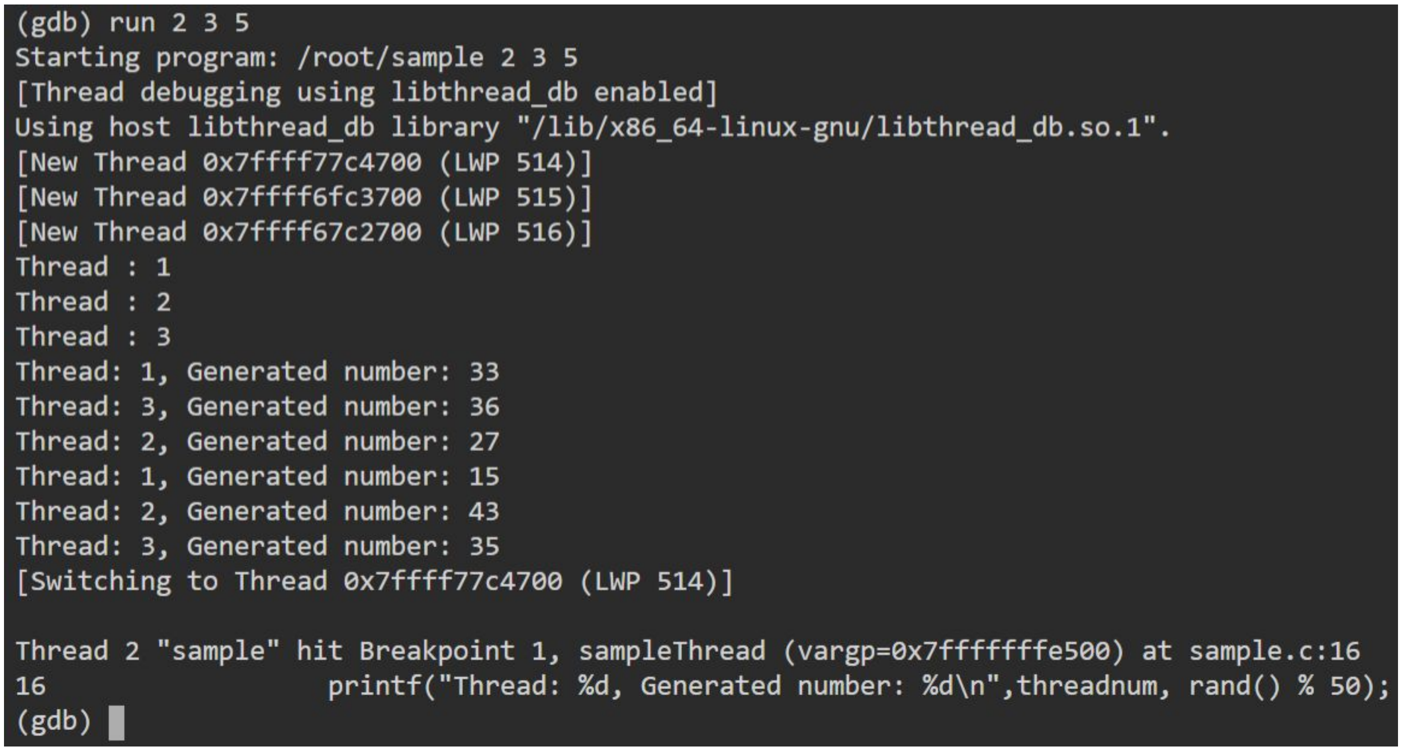
Command: info break



```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000000089d in sampleThread at sample.c:16
        stop only if i==2
```

Step 11: Run the program

Command: run 2 3 5



```
(gdb) run 2 3 5
Starting program: /root/sample 2 3 5
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff77c4700 (LWP 514)]
[New Thread 0x7ffff6fc3700 (LWP 515)]
[New Thread 0x7ffff67c2700 (LWP 516)]
Thread : 1
Thread : 2
Thread : 3
Thread: 1, Generated number: 33
Thread: 3, Generated number: 36
Thread: 2, Generated number: 27
Thread: 1, Generated number: 15
Thread: 2, Generated number: 43
Thread: 3, Generated number: 35
[Switching to Thread 0x7ffff77c4700 (LWP 514)]

Thread 2 "sample" hit Breakpoint 1, sampleThread (vargp=0x7fffffffe500) at sample.c:16
16                  printf("Thread: %d, Generated number: %d\n",threadnum, rand() % 50);
(gdb) 
```

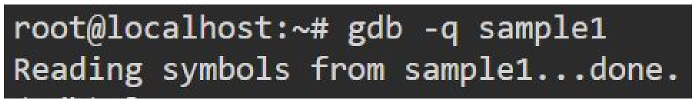
As one can observe, the control was passed through the breakpoint location multiple time but it only triggered when the condition turns out to be true.

Setting Temporary Breakpoints

Instead of setting permanent breakpoints that can will trigger every time and will need to be removed manually, the user can also go for temporary breakpoints that are enabled only for one stop and then get deleted automatically.

Step 12: Start GDB with the sample1 binary in quiet mode.

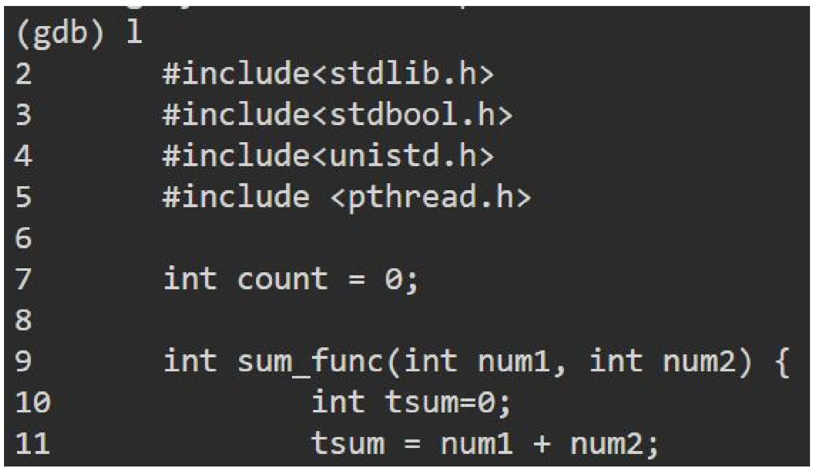
Command: gdb -q sample1



```
root@localhost:~# gdb -q sample1
Reading symbols from sample1...done.
```

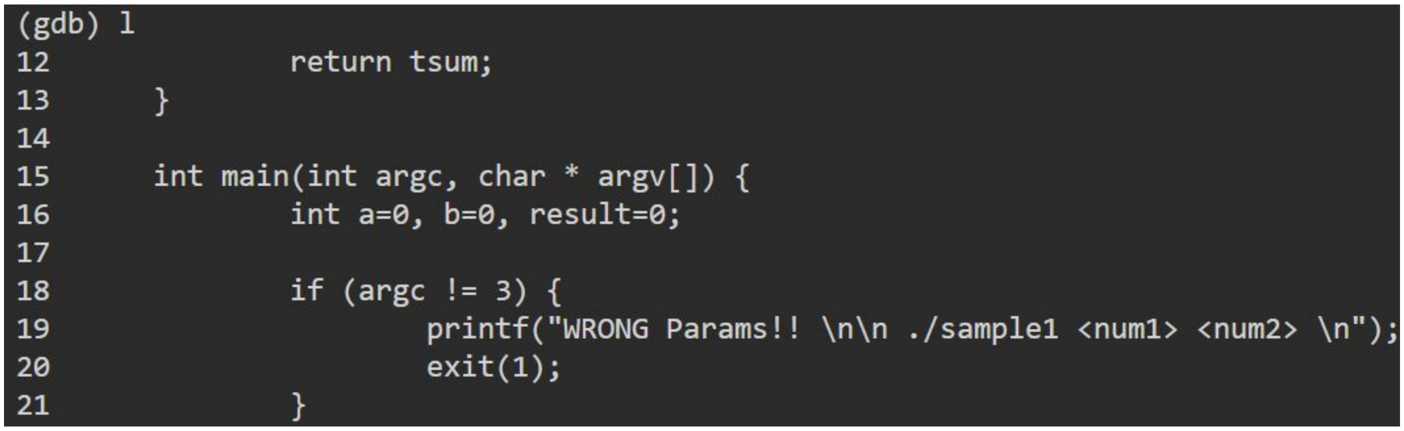
Step 13: Check the source code

Command: l (small case L)

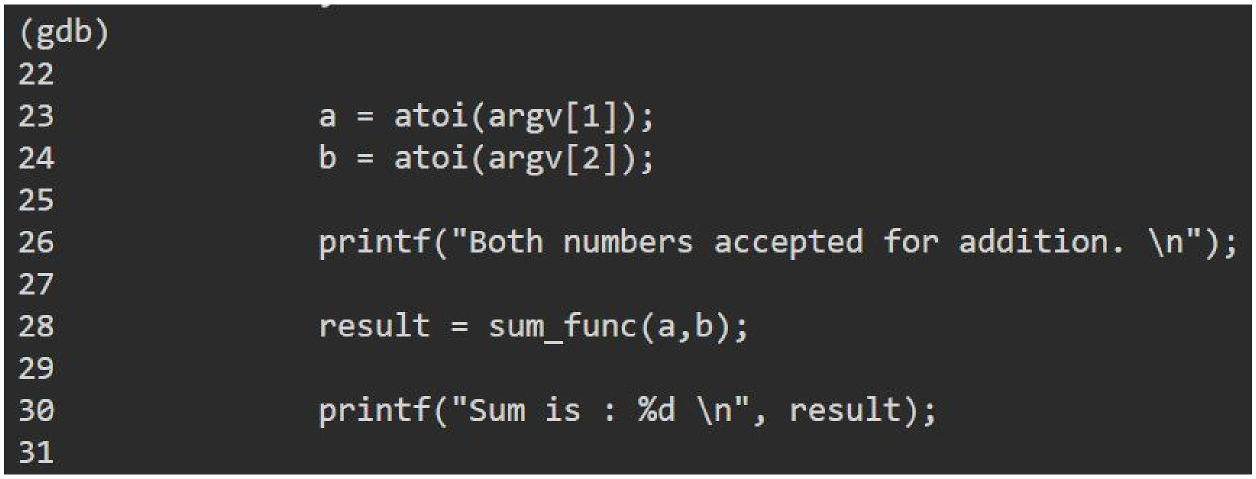


```
(gdb) l
2       #include<stdlib.h>
3       #include<stdbool.h>
4       #include<unistd.h>
5       #include <pthread.h>
6
7       int count = 0;
8
9       int sum_func(int num1, int num2) {
10              int tsum=0;
11              tsum = num1 + num2;
```

Pressing “enter” fires the last used command i.e. l (small case L) in this case.



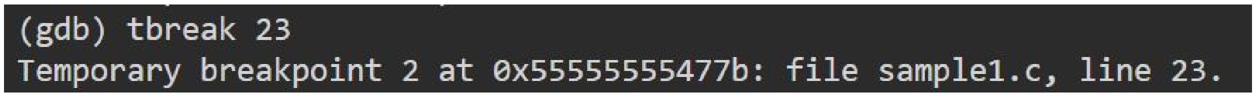
```
(gdb) l
12              return tsum;
13      }
14
15      int main(int argc, char * argv[]) {
16              int a=0, b=0, result=0;
17
18              if (argc != 3) {
19                      printf("WRONG Params!! \n\n ./sample1 <num1> <num2> \n");
20                      exit(1);
21              }
```



```
(gdb)
22
23              a = atoi(argv[1]);
24              b = atoi(argv[2]);
25
26              printf("Both numbers accepted for addition. \n");
27
28              result = sum_func(a,b);
29
30              printf("Sum is : %d \n", result);
31
```

Step 14: Set a temporary breakpoint on line number 23.

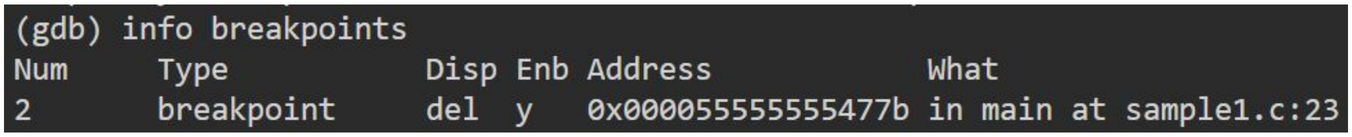
Command: tbreak 23



```
(gdb) tbreak 23
Temporary breakpoint 2 at 0x55555555477b: file sample1.c, line 23.
```

Step 15: Check the breakpoints

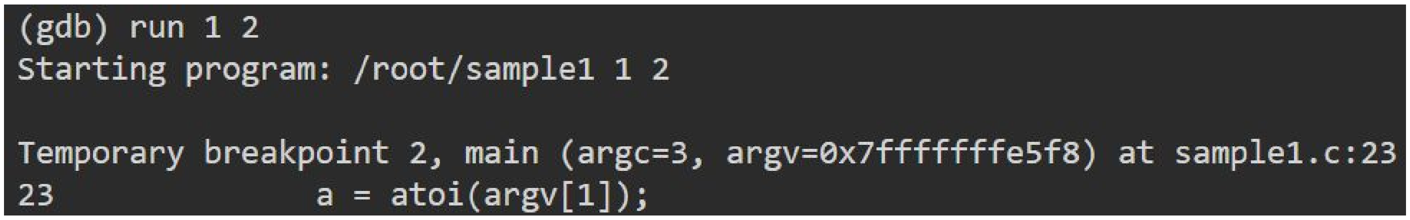
Command: info breakpoints



```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
2       breakpoint     del  y   0x000055555555477b in main at sample1.c:23
```

Step 16: Run the program

Command: run 1 2



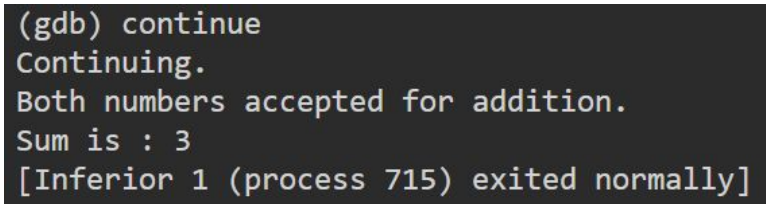
```
(gdb) run 1 2
Starting program: /root/sample1 1 2

Temporary breakpoint 2, main (argc=3, argv=0x7fffffffe5f8) at sample1.c:23
23              a = atoi(argv[1]);
```

The breakpoint is hit and the program execution has stopped at line 23.

Step 17: Continue with the program execution.

Command continue or c can be used to execute the program to the end or next breakpoint.

Command: continue

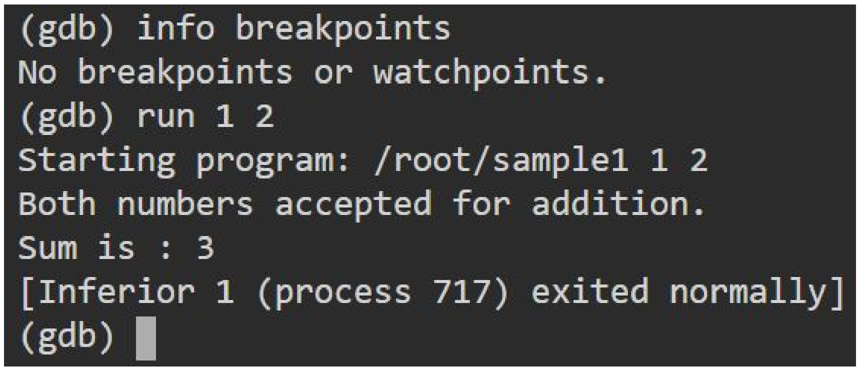


```
(gdb) continue
Continuing.
Both numbers accepted for addition.
Sum is : 3
[Inferior 1 (process 715) exited normally]
```

The program terminated.

Step 18: Check the breakpoint and

Commands:
info breakpoints
run 1 2



```
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb) run 1 2
Starting program: /root/sample1 1 2
Both numbers accepted for addition.
Sum is : 3
[Inferior 1 (process 717) exited normally]
(gdb) 
```

Setting Hardware Temporary Breakpoints

Step 19: GDB also allows the user to set up a hardware-assisted breakpoint. The args are the same as for the break command and the breakpoint is set in the same way, but the breakpoint requires hardware support and some target hardware may not have this support.

The main purpose of this kind of breakpoint is EPROM/ROM code debugging.

Unfortunately, the lab machine doesn't have support for this mode.

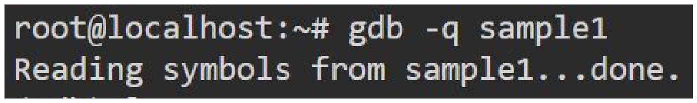
Example Command: hbreak 10

Set Regex based Breakpoints

GDB supports setting up a regex-based breakpoint that can be used to set breakpoints on functions matching the provided regex pattern.

Step 20: Start GDB with the sample1 binary in quiet mode.

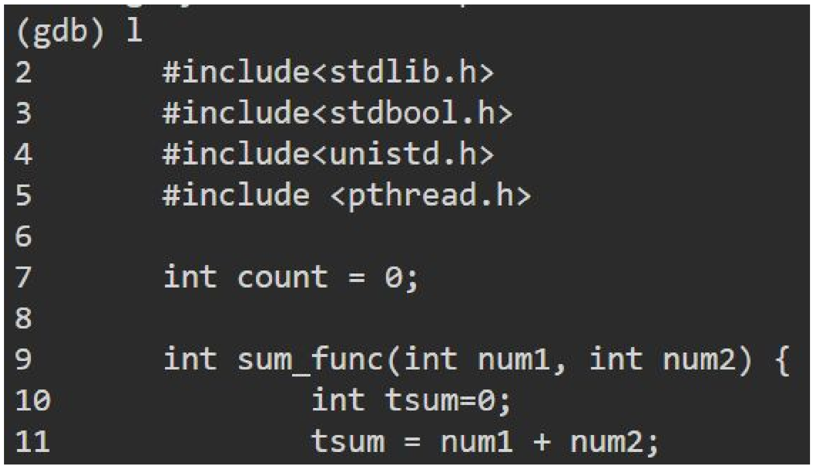
Command: gdb -q sample1



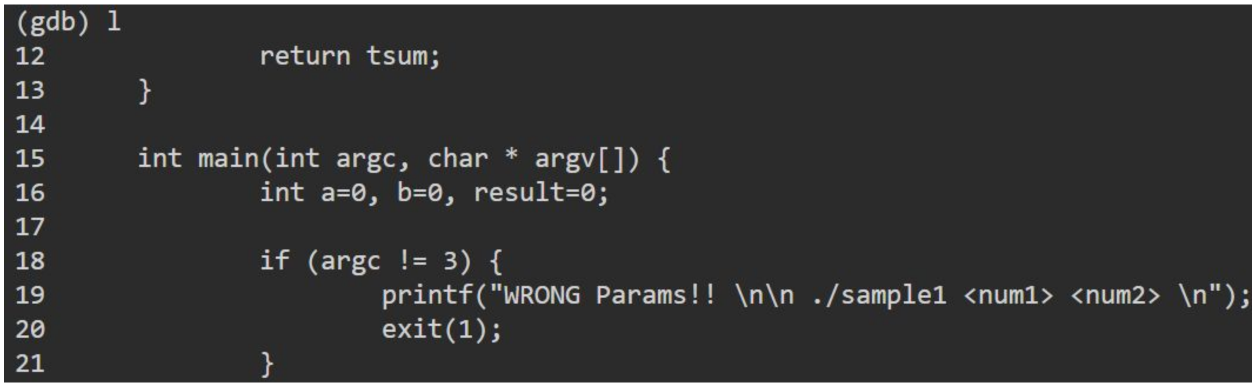
```
root@localhost:~# gdb -q sample1
Reading symbols from sample1...done.
```

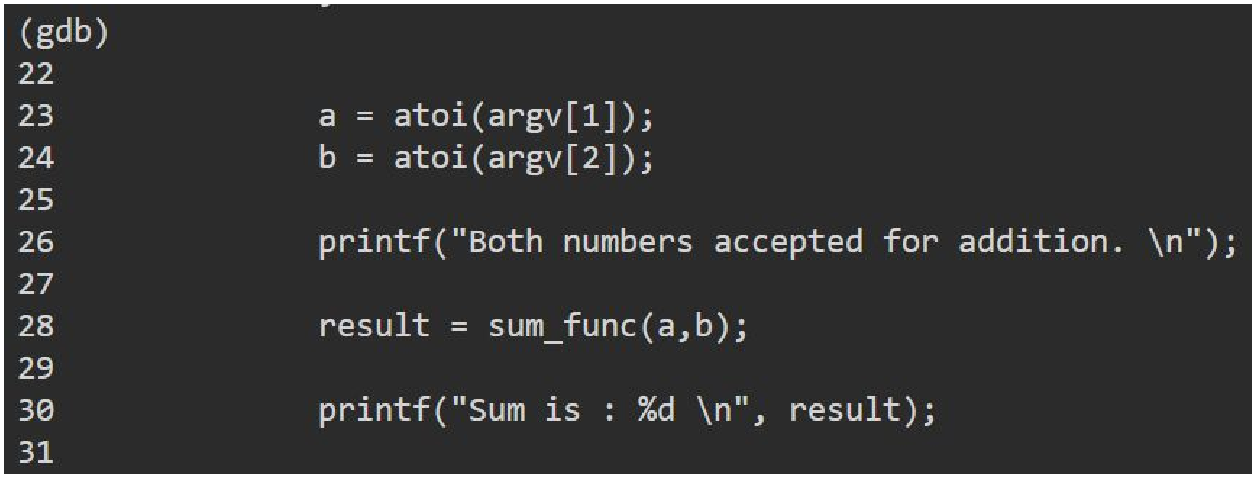
Step 1: Check the source code

Command: l   (small case L)


```
(gdb) l
2       #include<stdlib.h>
3       #include<stdbool.h>
4       #include<unistd.h>
5       #include <pthread.h>
6
7       int count = 0;
8
9       int sum_func(int num1, int num2) {
10              int tsum=0;
11              tsum = num1 + num2;
```

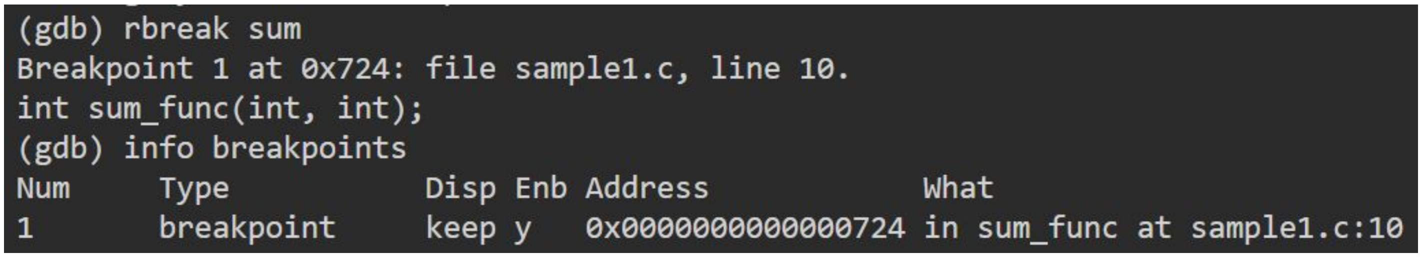
Pressing “enter” fires the last used command i.e. l (small case L) in this case.


```
(gdb) l
12              return tsum;
13      }
14
15      int main(int argc, char * argv[]) {
16              int a=0, b=0, result=0;
17
18              if (argc != 3) {
19                      printf("WRONG Params!! \n\n ./sample1 <num1> <num2> \n");
20                      exit(1);
21              }
```


```
(gdb)
22
23              a = atoi(argv[1]);
24              b = atoi(argv[2]);
25
26              printf("Both numbers accepted for addition. \n");
27
28              result = sum_func(a,b);
29
30              printf("Sum is : %d \n", result);
31
```

Step 21: Set breakpoints on all functions that have "sum" string in the function name. Then, check the list of breakpoints.

Commands:
rbreak sum
info breakpoint

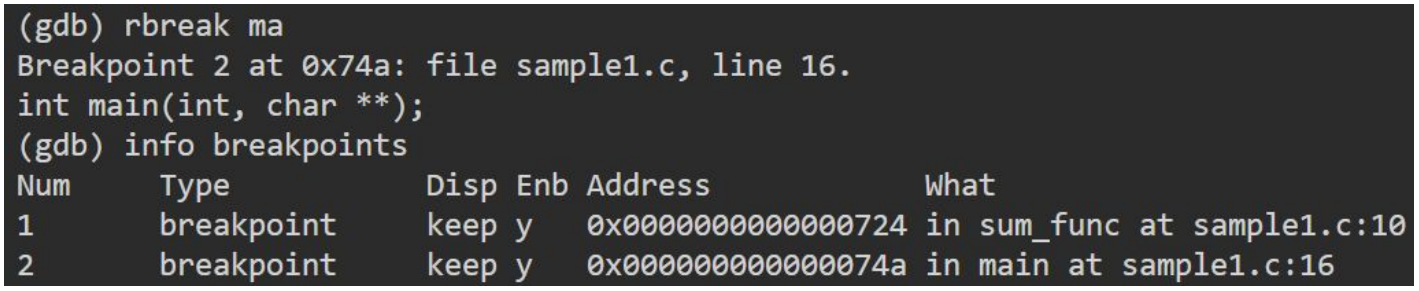


```
(gdb) rbreak sum
Breakpoint 1 at 0x724: file sample1.c, line 10.
int sum_func(int, int);
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000000724 in sum_func at sample1.c:10
```

The breakpoint was set on sum_func() function.

Step 22: Set breakpoints on all functions that have "ma" string in the function name. Then, check the list of breakpoints.

Commands:
rbreak ma
info breakpoint



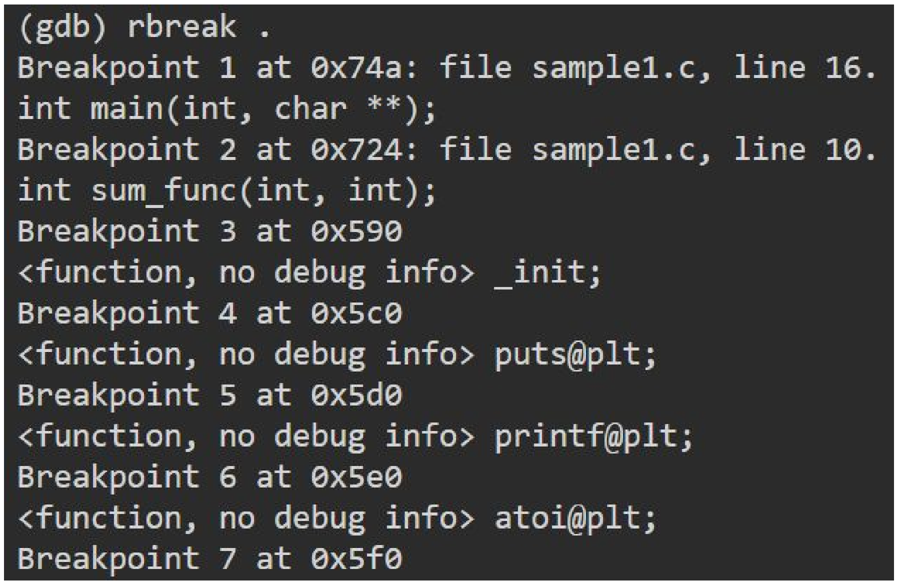
```
(gdb) rbreak ma
Breakpoint 2 at 0x74a: file sample1.c, line 16.
int main(int, char **);
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000000724 in sum_func at sample1.c:10
2       breakpoint     keep y   0x000000000000074a in main at sample1.c:16
```

The breakpoint was set on the main() function.

Note: The rbreak command assume a wildcard (*) before and after passed argument. Hence, it will work like grep *ma*. To only match the function name starting from a string sum, the argument will need to be ^sum

Step 23: Restart the GDB or remove all breakpoints. Set breakpoints on all functions of the program.

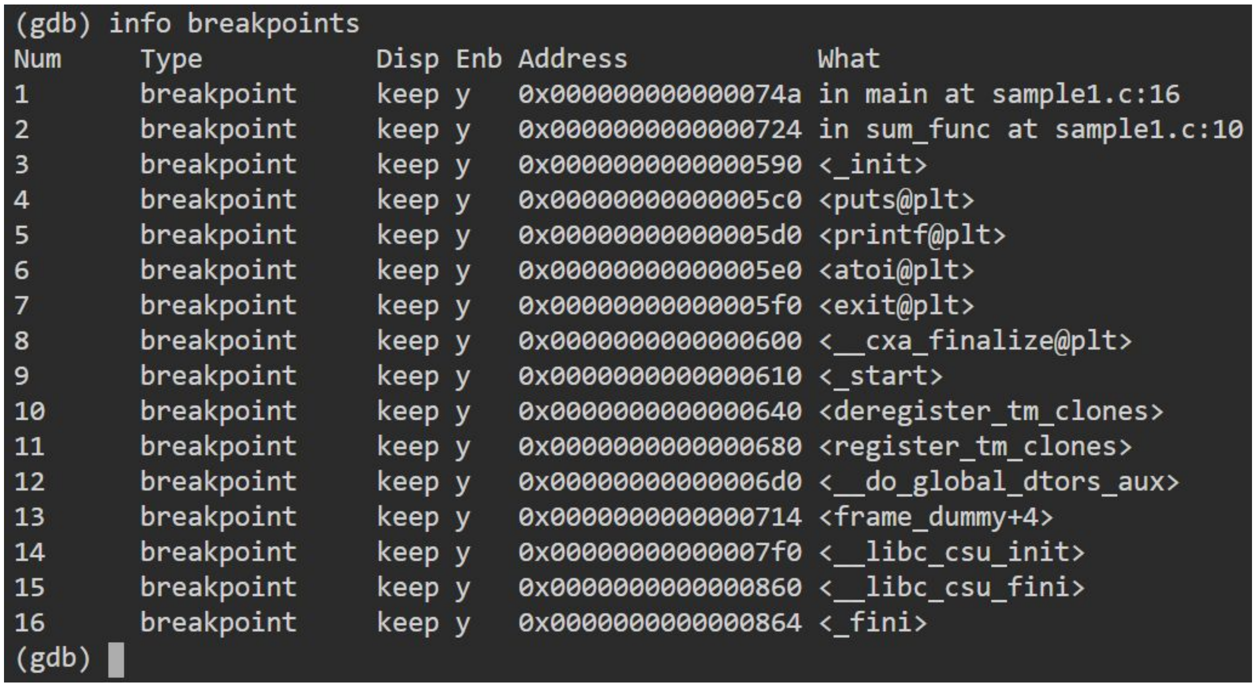
Command: rbreak .



```
(gdb) rbreak .
Breakpoint 1 at 0x74a: file sample1.c, line 16.
int main(int, char **);
Breakpoint 2 at 0x724: file sample1.c, line 10.
int sum_func(int, int);
Breakpoint 3 at 0x590
<function, no debug info> _init;
Breakpoint 4 at 0x5c0
<function, no debug info> puts@plt;
Breakpoint 5 at 0x5d0
<function, no debug info> printf@plt;
Breakpoint 6 at 0x5e0
<function, no debug info> atoi@plt;
Breakpoint 7 at 0x5f0
```

Step 24: Check the list of breakpoints.

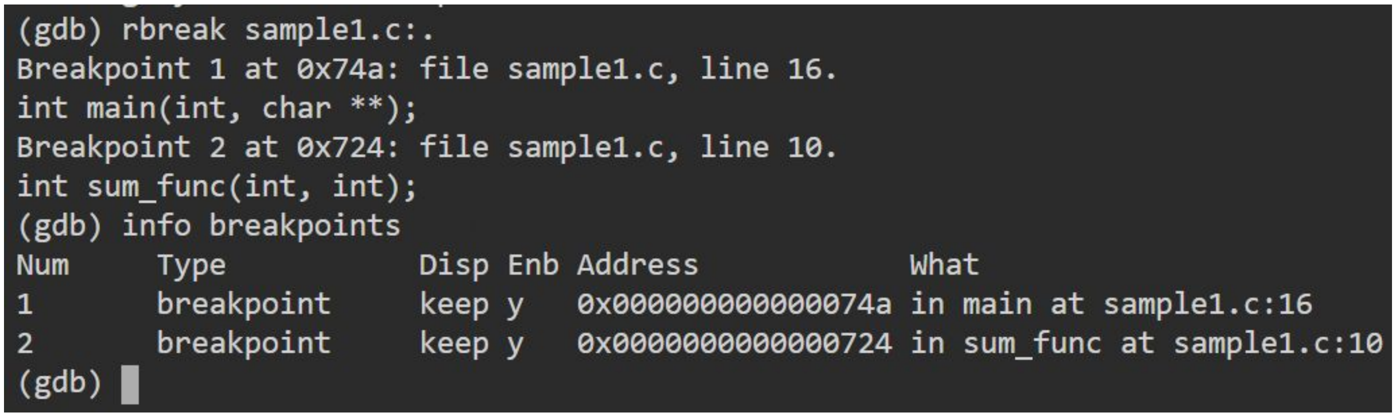
Command: info breakpoint


```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000000074a in main at sample1.c:16
2       breakpoint     keep y   0x0000000000000724 in sum_func at sample1.c:10
3       breakpoint     keep y   0x0000000000000590 <_init>
4       breakpoint     keep y   0x00000000000005c0 <puts@plt>
5       breakpoint     keep y   0x00000000000005d0 <printf@plt>
6       breakpoint     keep y   0x00000000000005e0 <atoi@plt>
7       breakpoint     keep y   0x00000000000005f0 <exit@plt>
8       breakpoint     keep y   0x0000000000000600 <__cxa_finalize@plt>
9       breakpoint     keep y   0x0000000000000610 <_start>
10      breakpoint     keep y   0x0000000000000640 <deregister_tm_clones>
11      breakpoint     keep y   0x0000000000000680 <register_tm_clones>
12      breakpoint     keep y   0x00000000000006d0 <__do_global_dtors_aux>
13      breakpoint     keep y   0x0000000000000714 <frame_dummy+4>
14      breakpoint     keep y   0x00000000000007f0 <__libc_csu_init>
15      breakpoint     keep y   0x0000000000000860 <__libc_csu_fini>
16      breakpoint     keep y   0x0000000000000864 <_fini>
(gdb) 
```

The breakpoints are set on all functions that are to be executed under this function. However, it also contains the library function which might not be of interest to the user.

Step 25: Set breakpoints on all functions present only in file sample1.c and check the list of breakpoints.

Commands:
rbreak sample1.c:.
info breakpoint

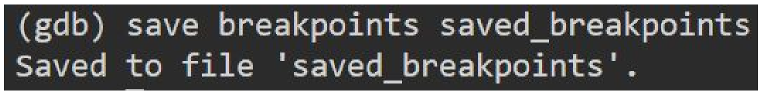

```
(gdb) rbreak sample1.c:.
Breakpoint 1 at 0x74a: file sample1.c, line 16.
int main(int, char **);
Breakpoint 2 at 0x724: file sample1.c, line 10.
int sum_func(int, int);
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000000074a in main at sample1.c:16
2       breakpoint     keep y   0x0000000000000724 in sum_func at sample1.c:10
(gdb) 
```

In this manner, one can set breakpoints on all functions of interest.

Save Breakpoints to File

The breakpoints can be saved in a file for future use.

Command: save breakpoints saved_breakpoints

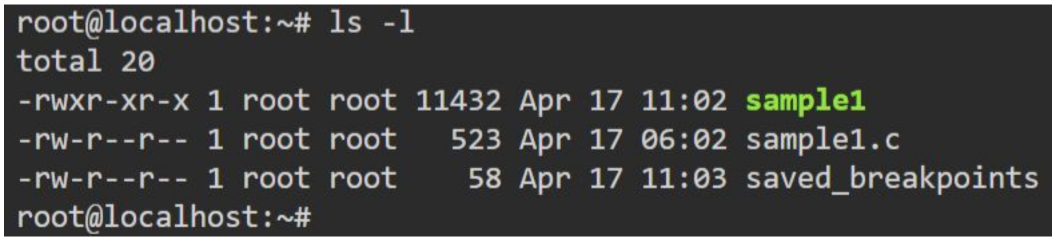


```
(gdb) save breakpoints saved_breakpoints
Saved to file 'saved_breakpoints'.
```

Note: Passing filename to the command is NOT optional.

Check the present working directory and verify the file has been created.

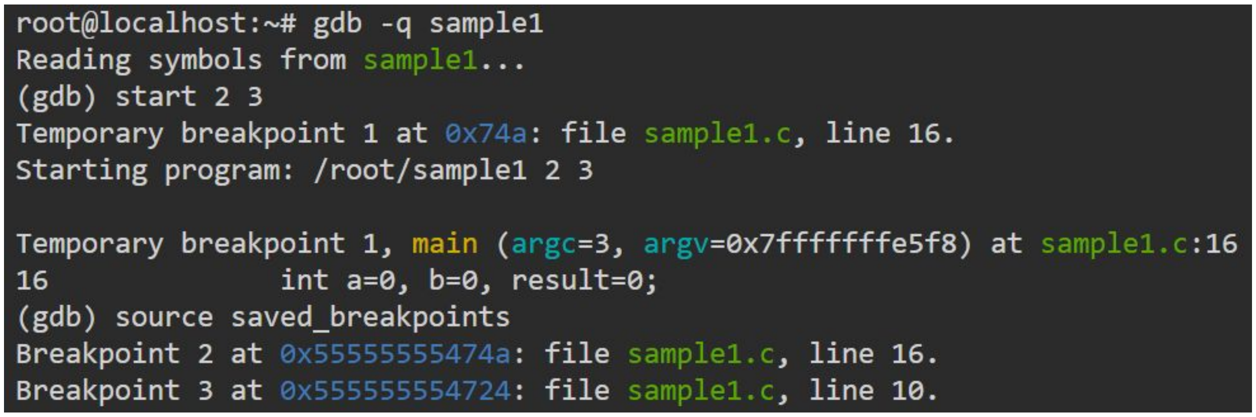
Command: ls -l



```
root@localhost:~# ls -l
total 20
-rwxr-xr-x 1 root root 11432 Apr 17 11:02 sample1
-rw-r--r-- 1 root root   523 Apr 17 06:02 sample1.c
-rw-r--r-- 1 root root    58 Apr 17 11:03 saved_breakpoints
root@localhost:~#
```

Restart the GDB with the sample1 binary and load the saved breakpoints from the file.

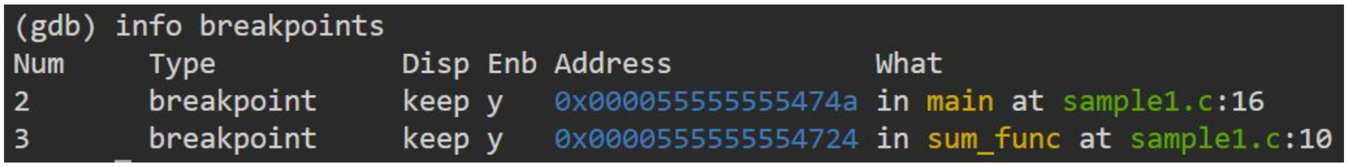
Commands:
gdb -q sample1
start 2 3
source saved_breakpoints


```
root@localhost:~# gdb -q sample1
Reading symbols from sample1...
(gdb) start 2 3
Temporary breakpoint 1 at 0x74a: file sample1.c, line 16.
Starting program: /root/sample1 2 3

Temporary breakpoint 1, main (argc=3, argv=0x7fffffffe5f8) at sample1.c:16
16              int a=0, b=0, result=0;
(gdb) source saved_breakpoints
Breakpoint 2 at 0x55555555474a: file sample1.c, line 16.
Breakpoint 3 at 0x555555554724: file sample1.c, line 10.
```

The breakpoints are loaded into the GDB.

List the breakpoints to verify the same.

Command: info breakpoints


```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x000055555555474a in main at sample1.c:16
3       breakpoint     keep y   0x0000555555554724 in sum_func at sample1.c:10
```

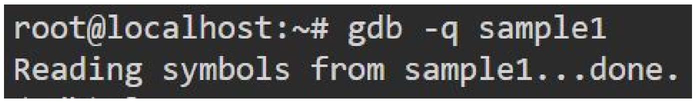
All breakpoints are loaded.

Setting Watchpoint

A watchpoint is used to halt execution whenever the value of expression changes. And, unlike breakpoint, in this case, the user doesn't have to even predict a location in program/code flow.

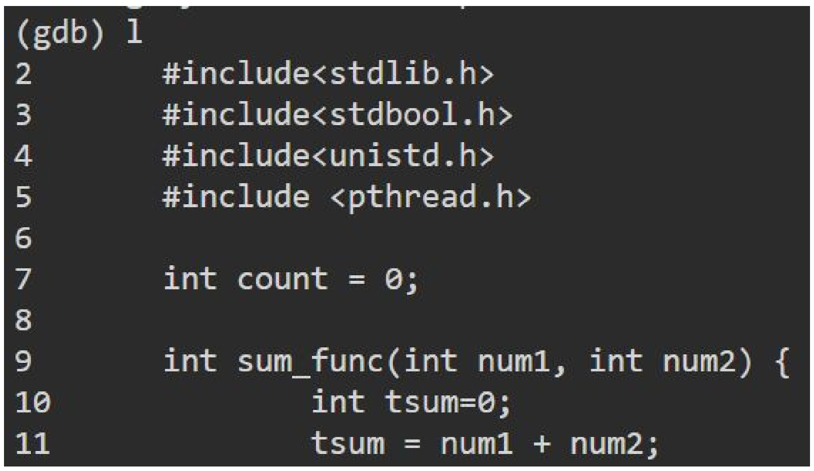
Step 26: Start GDB with the sample1 binary in quiet mode.

Command: gdb -q sample1


```
root@localhost:~# gdb -q sample1
Reading symbols from sample1...done.
```

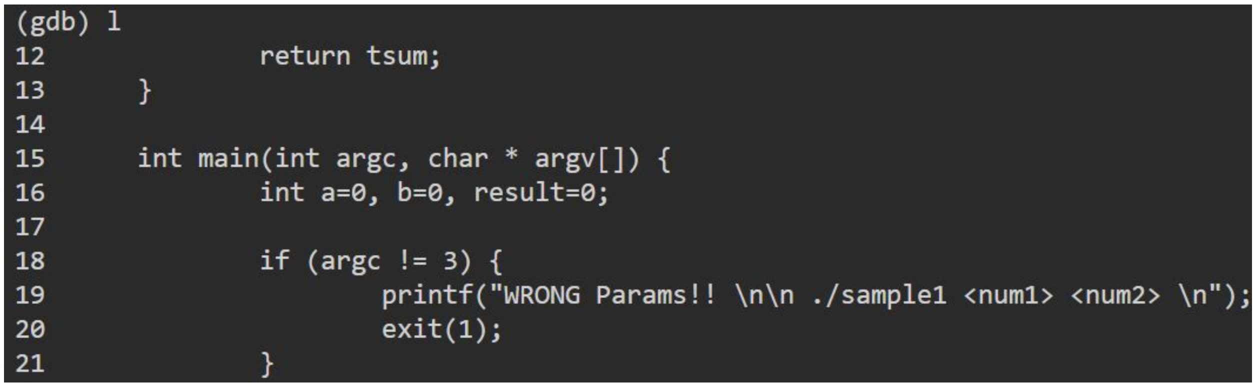
Step 27: Check the source code

Command: l (small case L)

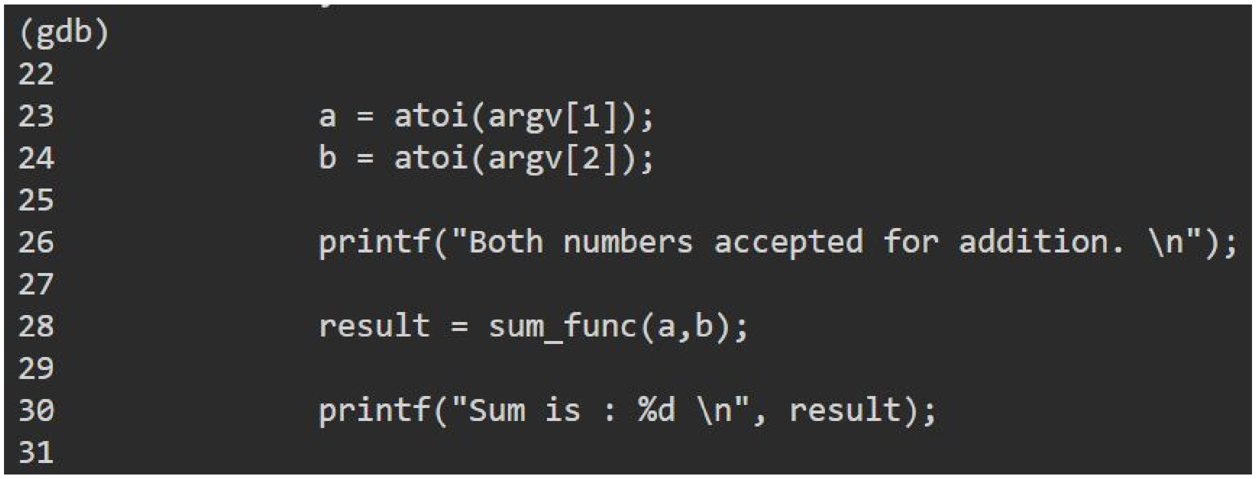


```
(gdb) l
2       #include<stdlib.h>
3       #include<stdbool.h>
4       #include<unistd.h>
5       #include <pthread.h>
6
7       int count = 0;
8
9       int sum_func(int num1, int num2) {
10              int tsum=0;
11              tsum = num1 + num2;
```

Pressing “enter” fires the last used command i.e. l (small case L) in this case.



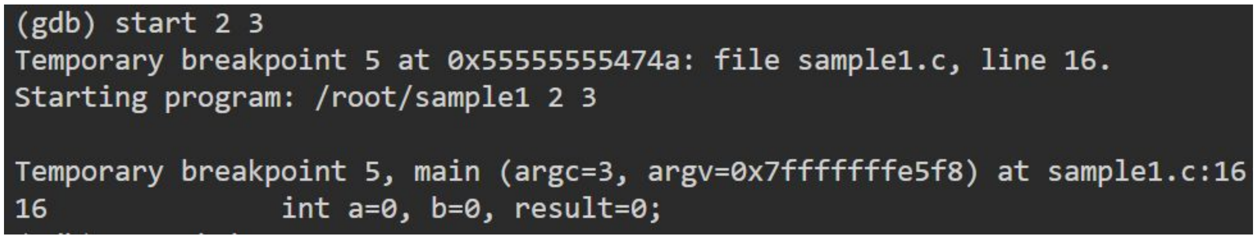
```
(gdb) l
12              return tsum;
13      }
14
15      int main(int argc, char * argv[]) {
16              int a=0, b=0, result=0;
17
18              if (argc != 3) {
19                      printf("WRONG Params!! \n\n ./sample1 <num1> <num2> \n");
20                      exit(1);
21              }
```



```
(gdb)
22
23              a = atoi(argv[1]);
24              b = atoi(argv[2]);
25
26              printf("Both numbers accepted for addition. \n");
27
28              result = sum_func(a,b);
29
30              printf("Sum is : %d \n", result);
31
```

Step 28: Start the program

Command: start 2 3

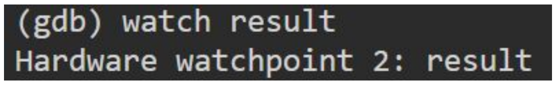


```
(gdb) start 2 3
Temporary breakpoint 5 at 0x55555555474a: file sample1.c, line 16.
Starting program: /root/sample1 2 3

Temporary breakpoint 5, main (argc=3, argv=0x7fffffffe5f8) at sample1.c:16
16              int a=0, b=0, result=0;
```

As expected, the execution stopped at main() function of the program.

Step 29: Set a watchpoint on variable “result”.

Command: watch result

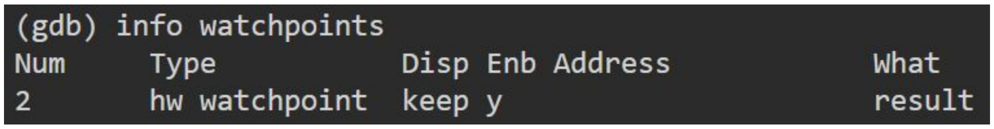


```
(gdb) watch result
Hardware watchpoint 2: result
```

Note: The watchpoints only work in the current context i.e. in the scope of the function being executed.

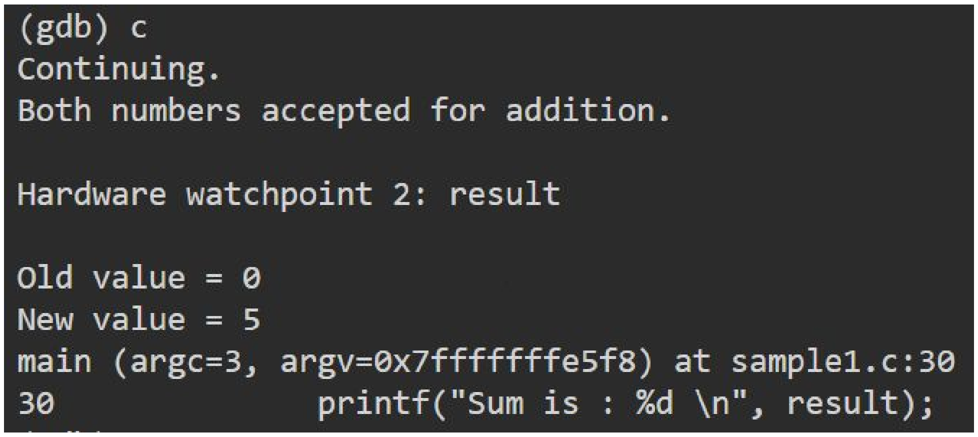
Step 30: List all watchpoints.

Command: info watchpoints


```
(gdb) info watchpoints
Num     Type           Disp Enb Address            What
2       hw watchpoint  keep y                      result
```

Step 31: Move from the breakpoint set on main() function.

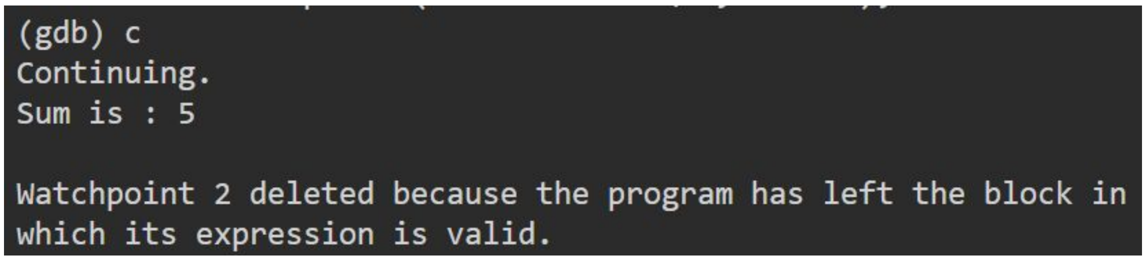
Command: c


```
(gdb) c
Continuing.
Both numbers accepted for addition.

Hardware watchpoint 2: result

Old value = 0
New value = 5
main (argc=3, argv=0x7fffffffe5f8) at sample1.c:30
30              printf("Sum is : %d \n", result);
```

The watchpoint was hit as the result variable is being used in the statement (line 30).

Step 32: Move from the breakpoint set on main() function.

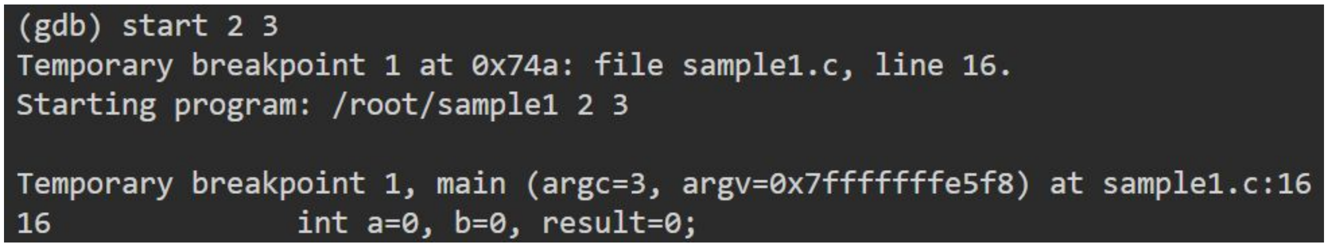
Command: c


```
(gdb) c
Continuing.
Sum is : 5

Watchpoint 2 deleted because the program has left the block in
which its expression is valid.
```

After this point, the program left the current function and hence the watchpoint got deleted.

Step 33: Run the program again.

Command: start 2 3

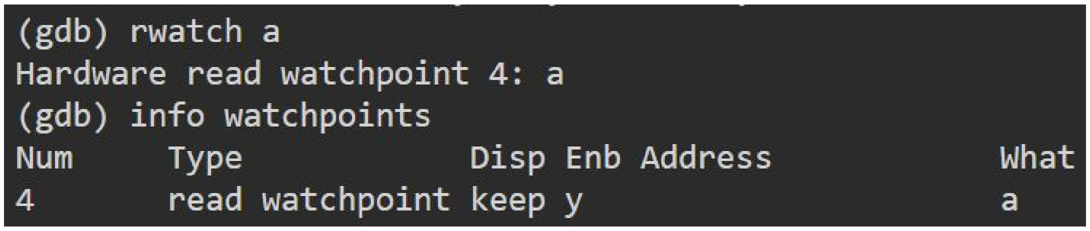


```
(gdb) start 2 3
Temporary breakpoint 1 at 0x74a: file sample1.c, line 16.
Starting program: /root/sample1 2 3

Temporary breakpoint 1, main (argc=3, argv=0x7fffffffe5f8) at sample1.c:16
16              int a=0, b=0, result=0;
```

As expected, the execution stopped at main() function of the program.

Step 34: Add a watchpoint for variable ‘a’ that is triggered when that variable is read. And, list the checkpoint.

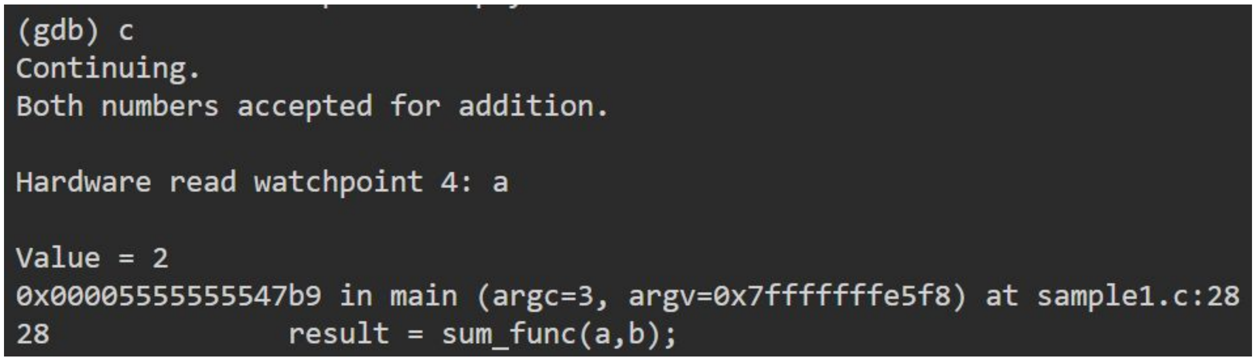
Commands:
rwatch a
info watchpoints



```
(gdb) rwatch a
Hardware read watchpoint 4: a
(gdb) info watchpoints
Num     Type            Disp Enb Address            What
4       read watchpoint keep y                      a
```

Step 35: Move from the breakpoint set on main() function.

Command: c

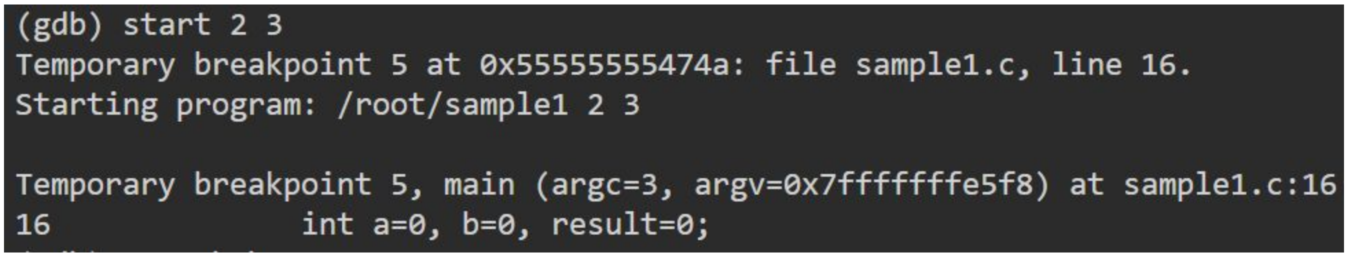


```
(gdb) c
Continuing.
Both numbers accepted for addition.

Hardware read watchpoint 4: a

Value = 2
0x00005555555547b9 in main (argc=3, argv=0x7fffffffe5f8) at sample1.c:28
28              result = sum_func(a,b);
```

The watchpoint was triggered when the a is passed as a parameter to another function.

Step 36: Run the program again.

Command: start 2 3

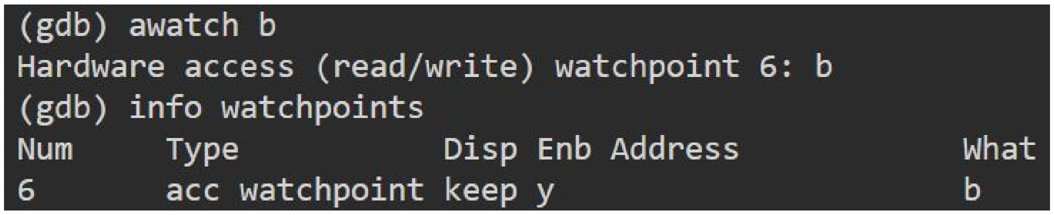


```
(gdb) start 2 3
Temporary breakpoint 5 at 0x55555555474a: file sample1.c, line 16.
Starting program: /root/sample1 2 3

Temporary breakpoint 5, main (argc=3, argv=0x7fffffffe5f8) at sample1.c:16
16              int a=0, b=0, result=0;
```

As expected, the execution stopped at main() function of the program.

Step 37: Add a watchpoint for variable ‘b’ that is triggered when that variable is either read from or written into by the program. And, also check the checkpoint list.

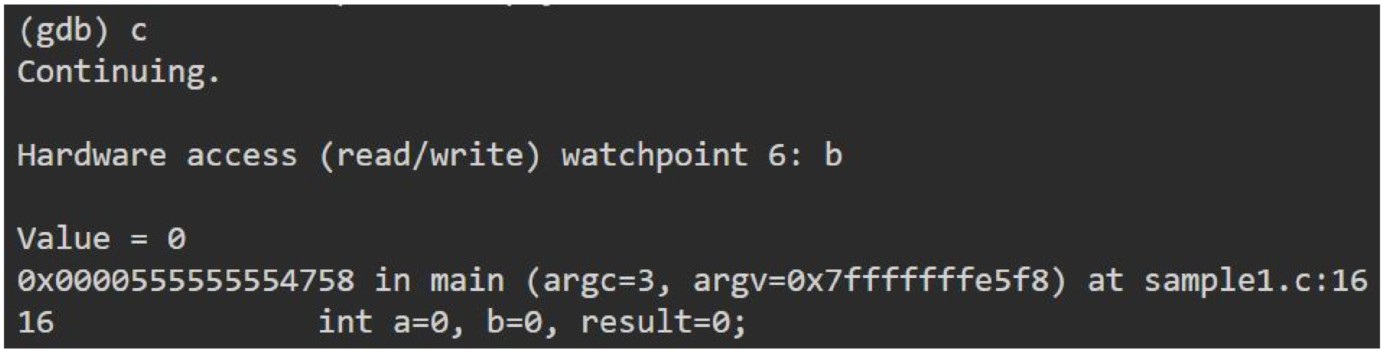
Commands:
awatch b
info watchpoints



```
(gdb) awatch b
Hardware access (read/write) watchpoint 6: b
(gdb) info watchpoints
Num     Type           Disp Enb Address            What
6       acc watchpoint keep y                      b
```

Step 38: Move from the breakpoint set on main() function.

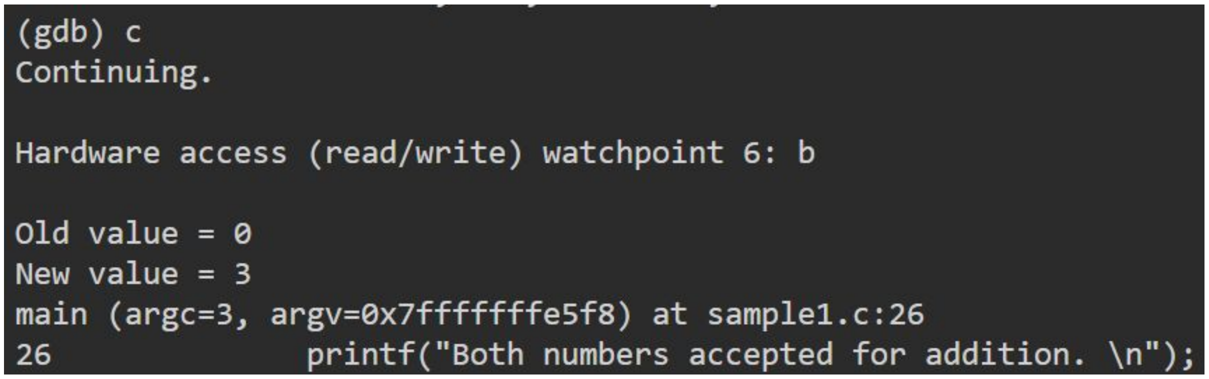
Command: c


```
(gdb) c
Continuing.

Hardware access (read/write) watchpoint 6: b

Value = 0
0x0000555555554758 in main (argc=3, argv=0x7fffffffe5f8) at sample1.c:16
16              int a=0, b=0, result=0;
```

The value is b is being initialized in line 16.

Step 39: Continue to the next break/watchpoint.

Command: c

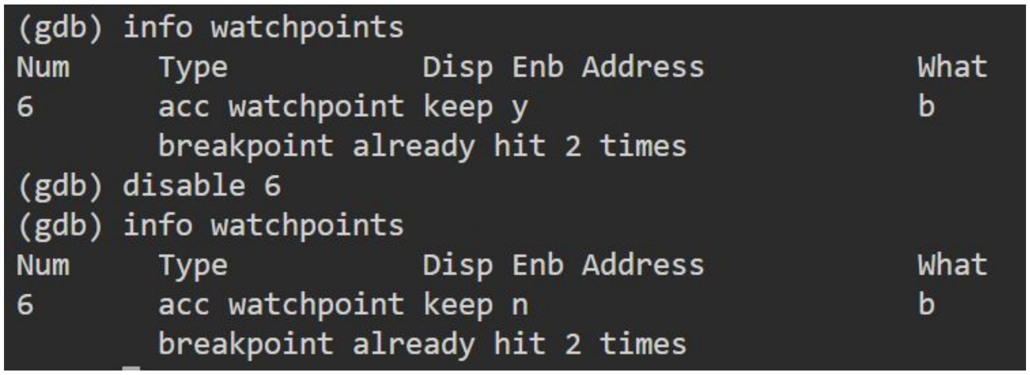

```
(gdb) c
Continuing.

Hardware access (read/write) watchpoint 6: b

Old value = 0
New value = 3
main (argc=3, argv=0x7fffffffe5f8) at sample1.c:26
26              printf("Both numbers accepted for addition. \n");
```

The next watchpoint is set when the passed value is being assigned to the variable b.

<u>Disabling Watchpoint/Breakpoints</u>

Step 40: The watchpoints can be disabled when not in use. List all watchpoints, disable the active watchpoints and again verify the change.

Commands:
info watchpoints
Disable 6
info watchpoints



```
(gdb) info watchpoints
Num     Type            Disp Enb Address            What
6       acc watchpoint  keep y                      b
        breakpoint already hit 2 times
(gdb) disable 6
(gdb) info watchpoints
Num     Type            Disp Enb Address            What
6       acc watchpoint  keep n                      b
        breakpoint already hit 2 times
```

Notice the value in column "Enb".

Step 41: Similarly, watchpoints can be enabled again using 'enable' command.

Example 1: Enable breakpoint number 6.

Command: enable 6

Example 2: Enable breakpoint numbers 7,8,9,10.

Command: enable 7,8,9,10

Example 3: Enable breakpoint number 4 to run once

Command: enable once 6

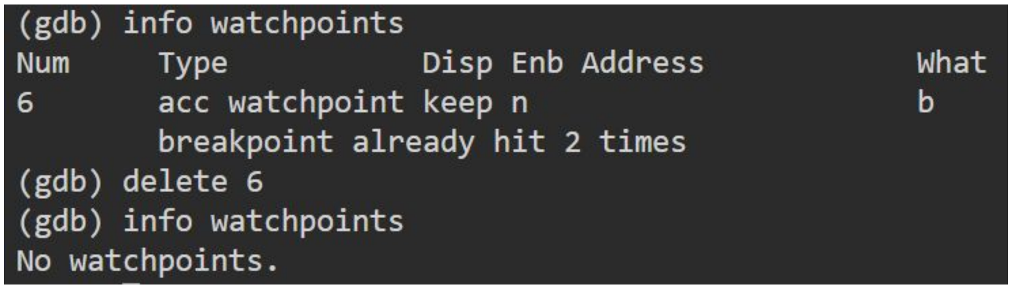
Example 4: Enable next 4 breakpoints

Command: enable count 4

<u>Deleting Watchpoint/Breakpoint</u>

Step 41: The watchpoints can be deleted once the objective is met. List all watchpoints, disable the active watchpoints and again verify the change.

Commands:
info watchpoints
Disable 6
info watchpoints



```
(gdb) info watchpoints
Num     Type           Disp Enb Address            What
6       acc watchpoint keep n                      b
        breakpoint already hit 2 times
(gdb) delete 6
(gdb) info watchpoints
No watchpoints.
```

The watchpoint will disappear from the list post deletion.

The same can be achieved by using the “clear” command.

Step 43: Clear the breakpoint set on the next step.

Command: clear

Similarly, to clear the breakpoint on a location, the location can be passed to clear command.

Example 1: To clear a breakpoint at line 16

Command: clear 16

Example 2: To clear a breakpoint at function sum()

Command: clear sum

In the case of multiple files, one can define the file name before the location.

Example: To clear a breakpoint at line 18 of sample.c file

Command: clear sample.c 18

Setting Catchpoint

The catchpoints are used to halt the execution in certain kinds of program events, such as C++ exceptions or the loading of a shared library.

Continuing

Continuing refers to resuming program execution until the program terminates normally (or a breakpoint is encountered).

The continue command (also abbreviated as c) is used for this.

Example 1: Continue till the next breakpoint or end of the program.

Command: continue or **Command:** c or **Command:** fg

Example 2: Continue till the next breakpoint or end of the program while ignoring a breakpoint 5 times at the current location.

Command: continue 5 or **Command:** c 5 or **Command:** fg 5

Note: This command is only useful when the program execution is already stopped on a breakpoint.

Stepping

Stepping refers to executing just one more "step" of the program after the execution is halted due to a breakpoint. The "step" here can be either one line of source code or one machine instruction.

step and next command

The step command (abbreviated as s) and the next command (abbreviated as n) are used to step one line of the program.

Example 1: Run the program until control reaches the next source line.

Command: step or **Command:** s or **Command:** next or **Command:** n

Example 2: Run the program for the next 5 lines and then halt the execution.

Command: step 5 or **Command:** s 5 or **Command:** next 5 or **Command:** n 5

stepi and nexti commands

The stepi command (abbreviated as si) and the nexti command (abbreviated as ni) are used to step one machine instruction of the program.

Syntax: The syntax is the same as step and next commands.

Difference between step and next

Step instruction steps into the function when it encounters a call to the function.

Next instruction steps over the function (doesn't enter the function) when it encounters a call to the function.

Running program till a milestone

until command

The until (or u) command runs the program until either the specified location or end of the current stack frame. Location can be line number, function name, etc.

Example: Stop the execution at line number 16

Command: until 16 or **Command:** u 16

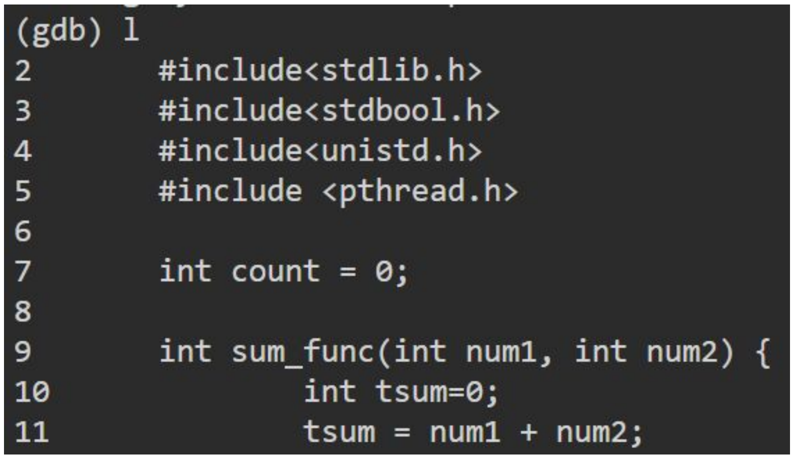
until command (without argument)

If no location is provided, runs the program to the next source line (like next command). However, it can skip the loop interaction as on encountering a jump, it automatically continues execution until the program counter is greater than the address of the jump.

Command: unti

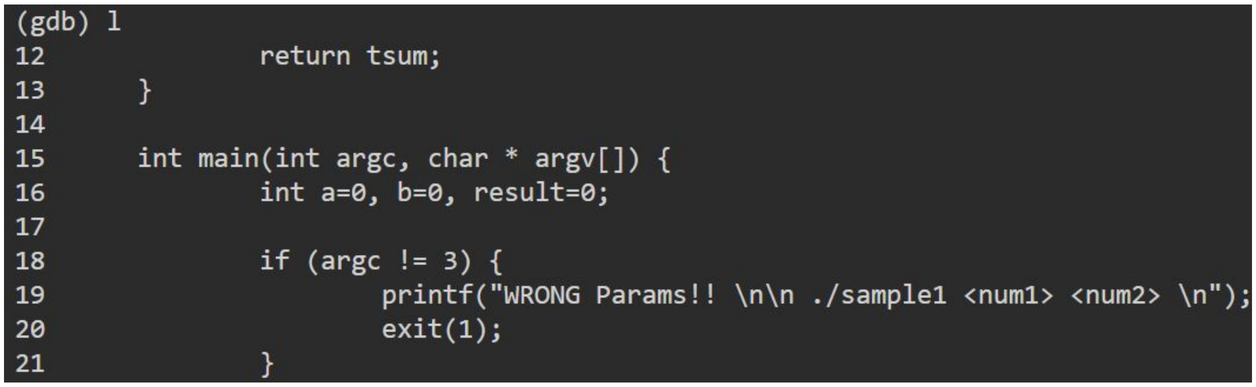
Step 45: Check the source code

Command: l (small case L)

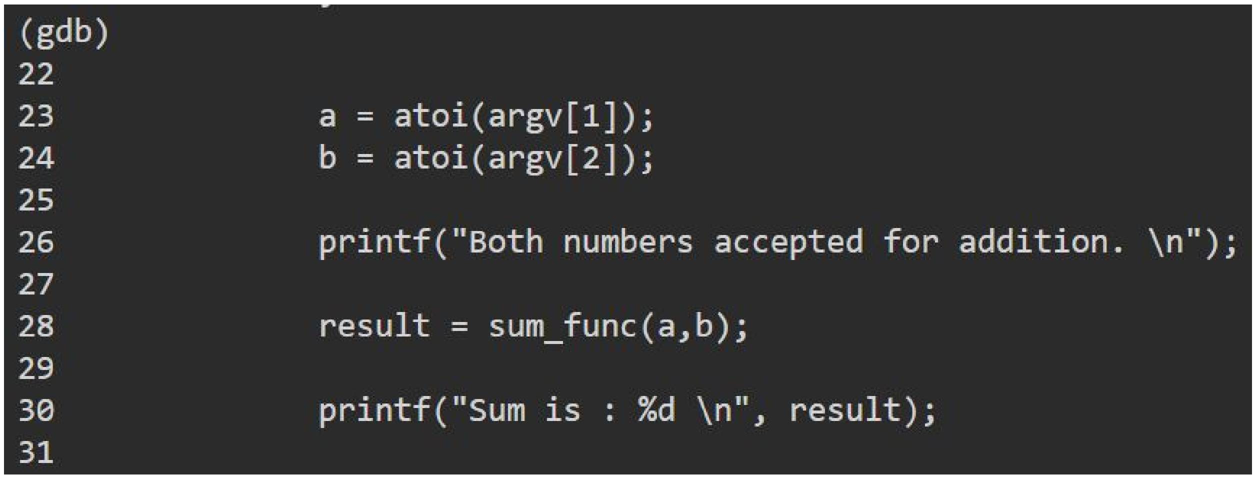


```
(gdb) l
2       #include<stdlib.h>
3       #include<stdbool.h>
4       #include<unistd.h>
5       #include <pthread.h>
6
7       int count = 0;
8
9       int sum_func(int num1, int num2) {
10              int tsum=0;
11              tsum = num1 + num2;
```

Pressing “enter” fires the last used command i.e. l (small case L) in this case.



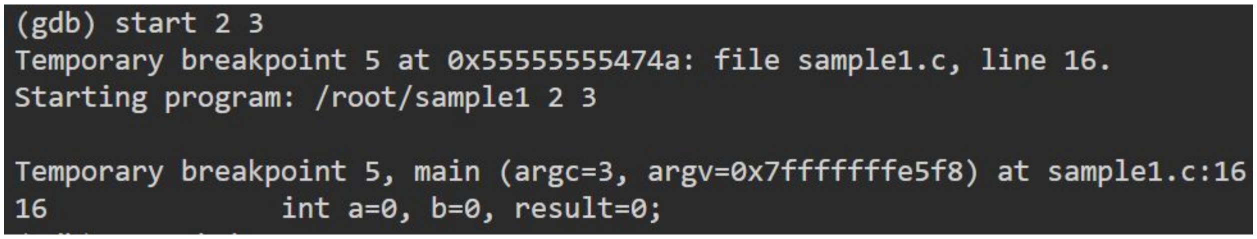
```
(gdb) l
12              return tsum;
13      }
14
15      int main(int argc, char * argv[]) {
16              int a=0, b=0, result=0;
17
18              if (argc != 3) {
19                      printf("WRONG Params!! \n\n ./sample1 <num1> <num2> \n");
20                      exit(1);
21              }
```



```
(gdb)
22
23              a = atoi(argv[1]);
24              b = atoi(argv[2]);
25
26              printf("Both numbers accepted for addition. \n");
27
28              result = sum_func(a,b);
29
30              printf("Sum is : %d \n", result);
31
```

Step 46: Start the program

Command: start 2 3

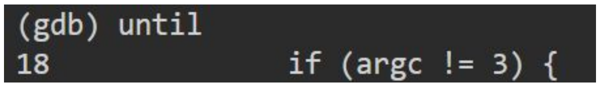


```
(gdb) start 2 3
Temporary breakpoint 5 at 0x55555555474a: file sample1.c, line 16.
Starting program: /root/sample1 2 3

Temporary breakpoint 5, main (argc=3, argv=0x7fffffffe5f8) at sample1.c:16
16              int a=0, b=0, result=0;
```

As expected, the execution stopped at the main() function of the program.

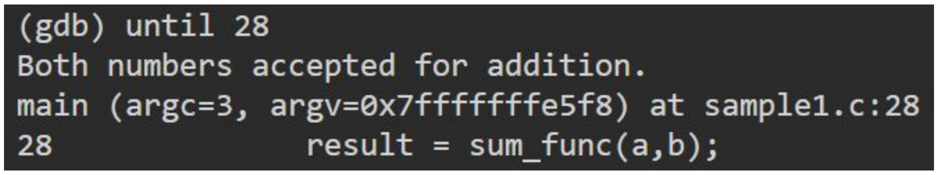
Step 47: Skip to the next line of the code using until command.

Command: until


```
(gdb) until
18              if (argc != 3) {
```

Step 48: Skip to code line number 28 using until command.

Command: until 28


```
(gdb) until 28
Both numbers accepted for addition.
main (argc=3, argv=0x7fffffffe5f8) at sample1.c:28
28              result = sum_func(a,b);
```

advance command

The advance command runs the program until either the specified location (like until command with a location) but it won't skip over recursive function calls and can run to locations target location out of the current stack frame.

Example: Stop the execution at line number 30

Command: advance 30

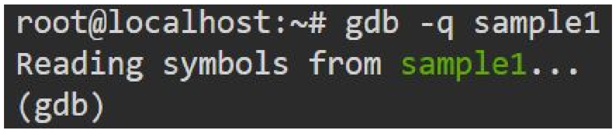
Dynamic Printf

Dynamic printf allows the user to print a log or values at a location during program execution. The main benefit of this approach is that it saves the effort of putting a print statement in the code, compile it and then load it in GDB again.

By default, the dynamic printf (or dprintf) command passes the arguments to printf for printing.

Step 49: Start the GDB with sample1 binary.

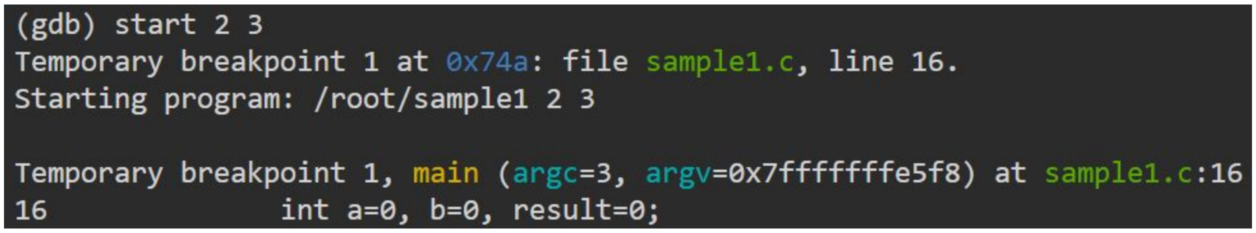
Command: gdb -q sample1



```
root@localhost:~# gdb -q sample1
Reading symbols from sample1...
(gdb)
```

Step 50: Start the program and pass it arguments

Command: start 2 3

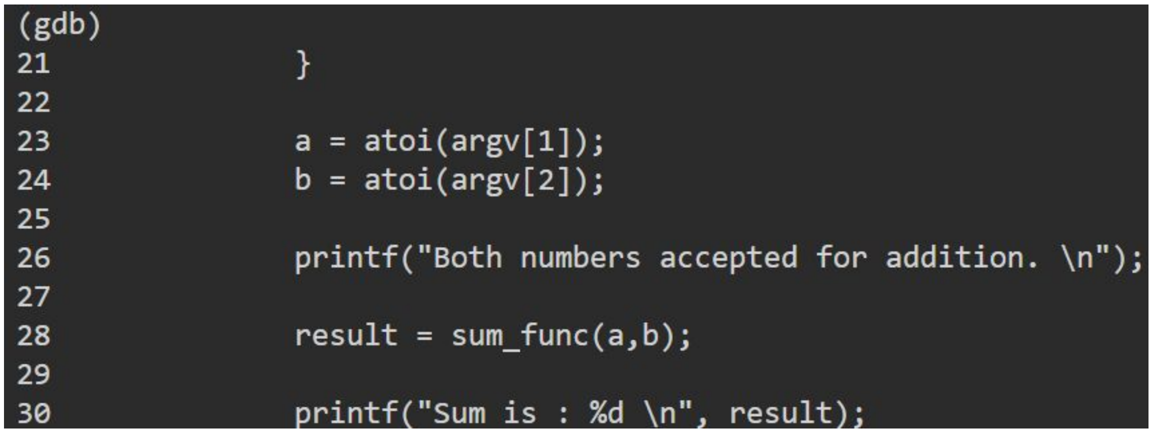


```
(gdb) start 2 3
Temporary breakpoint 1 at 0x74a: file sample1.c, line 16.
Starting program: /root/sample1 2 3

Temporary breakpoint 1, main (argc=3, argv=0x7fffffffe5f8) at sample1.c:16
16              int a=0, b=0, result=0;
```

The program execution stopped at the first line of the main() function.

Step 51: Check the source code of the binary.

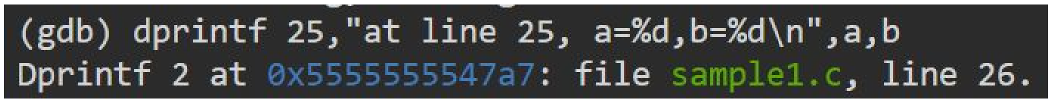
Command: l (Small case L)



```
(gdb)
21                      }
22
23                      a = atoi(argv[1]);
24                      b = atoi(argv[2]);
25
26                      printf("Both numbers accepted for addition. \n");
27
28                      result = sum_func(a,b);
29
30                      printf("Sum is : %d \n", result);
```

Step 52: Insert a dprintf statement to print the values of variables a and b at code line 25.

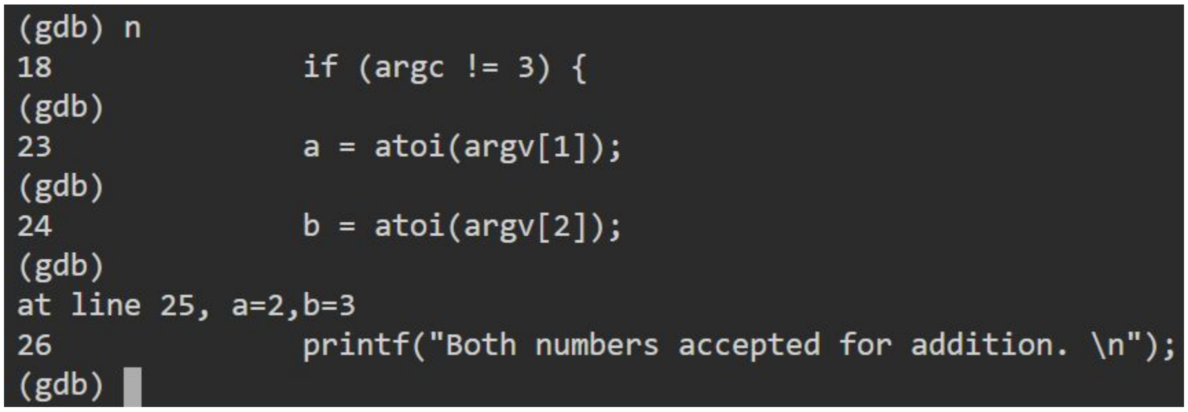
Command: dprintf 25,”at line 25,a=%d,b=%d\n”,a,b



```
(gdb) dprintf 25,"at line 25, a=%d,b=%d\n",a,b
Dprintf 2 at 0x5555555547a7: file sample1.c, line 26.
```

Step 53: Step to next instruction using "next" or "n" command till the time dprintf statement is not executed.

Command: n



```
(gdb) n
18              if (argc != 3) {
(gdb)
23              a = atoi(argv[1]);
(gdb)
24              b = atoi(argv[2]);
(gdb)
at line 25, a=2,b=3
26              printf("Both numbers accepted for addition. \n");
(gdb)
```

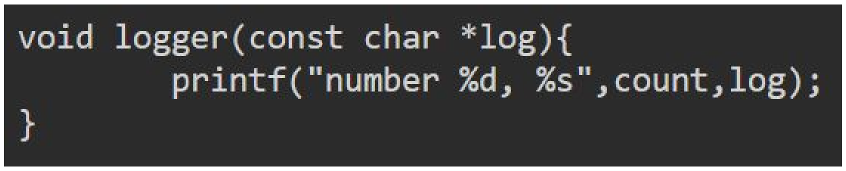
The dprintf output can be observed on the console.

By default, the dynamic printf (or dprintf) command passes the arguments to printf for printing. However, one can choose to use a custom function to print/handle the log.

Step 54: Add a new function logger() to the sample1.c file.

Code added:

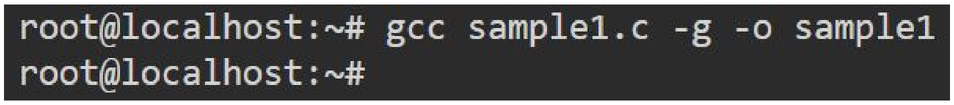
```
void logger(const char *log){
        printf("number %d, %s",count,log);
}
```



```
void logger(const char *log){
        printf("number %d, %s",count,log);
}
```

Step 55: Compile sample1.c file and enable debugging symbols.

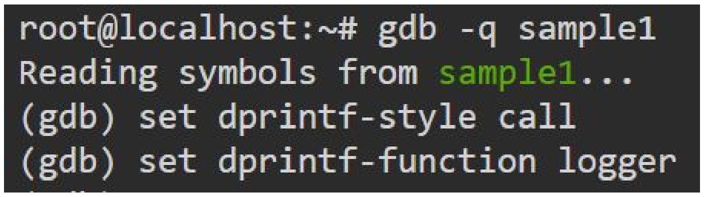
Command: gcc sample1.c -g -o sample1



```
root@localhost:~# gcc sample1.c -g -o sample1
root@localhost:~#
```

Step 56: Open GDB with sample1 binary. Select the “call” dprintf-style and define logger() function as target function.

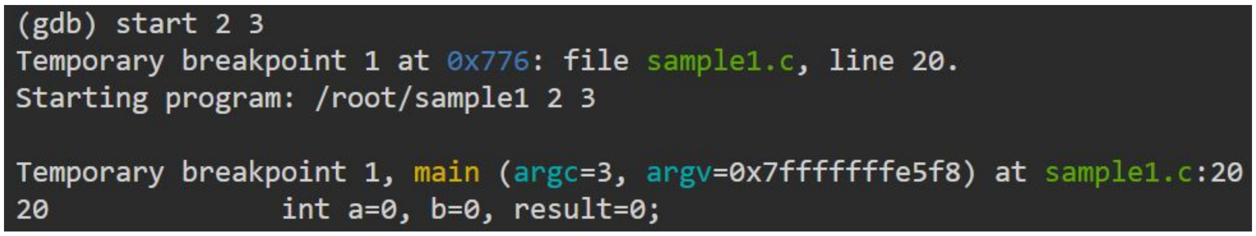
Commands:
gdb -q sample1
set dprintf-style call
set dprintf-function logger



```
root@localhost:~# gdb -q sample1
Reading symbols from sample1...
(gdb) set dprintf-style call
(gdb) set dprintf-function logger
```

Step 57: Start the program and pass its arguments.

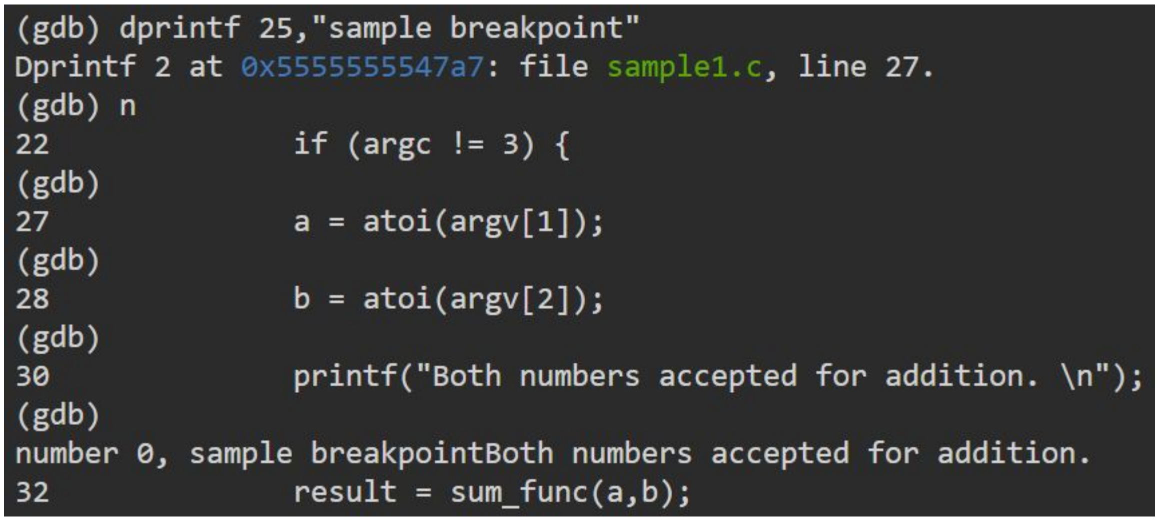
Command: start 2 3



```
(gdb) start 2 3
Temporary breakpoint 1 at 0x776: file sample1.c, line 20.
Starting program: /root/sample1 2 3

Temporary breakpoint 1, main (argc=3, argv=0x7fffffffe5f8) at sample1.c:20
20              int a=0, b=0, result=0;
```

Step 58: Set the dprintf on line 25, step to the next instruction using the “next” or “n” command till the dprintf statement gets executed.

Commands:
Dprintf 25,”sample breakpoint”
n



```
(gdb) dprintf 25,"sample breakpoint"
Dprintf 2 at 0x5555555547a7: file sample1.c, line 27.
(gdb) n
22              if (argc != 3) {
(gdb)
27              a = atoi(argv[1]);
(gdb)
28              b = atoi(argv[2]);
(gdb)
30              printf("Both numbers accepted for addition. \n");
(gdb)
number 0, sample breakpointBoth numbers accepted for addition.
32              result = sum_func(a,b);
```

In this manner, dprintf statement can be used to check the values during debugging with GDB.

References:

1. GDB Documentation (https://sourceware.org/gdb/current/onlinedocs/gdb)