Exploring GDB commands

Assignment-2

GROUP- 7

MEMBERS:

1. Dipayan Maity	2021CSB039
2. Aaratrika Banerjee	2021CSB040
3. Pantho Propan Debnath	2021CSB041
4. Dipmay Biswas	2021CSB043
5. Ketan Khandenwal	2021CSB045

❖ Write a short note on the GNU debugger(GDB) command in Linux.

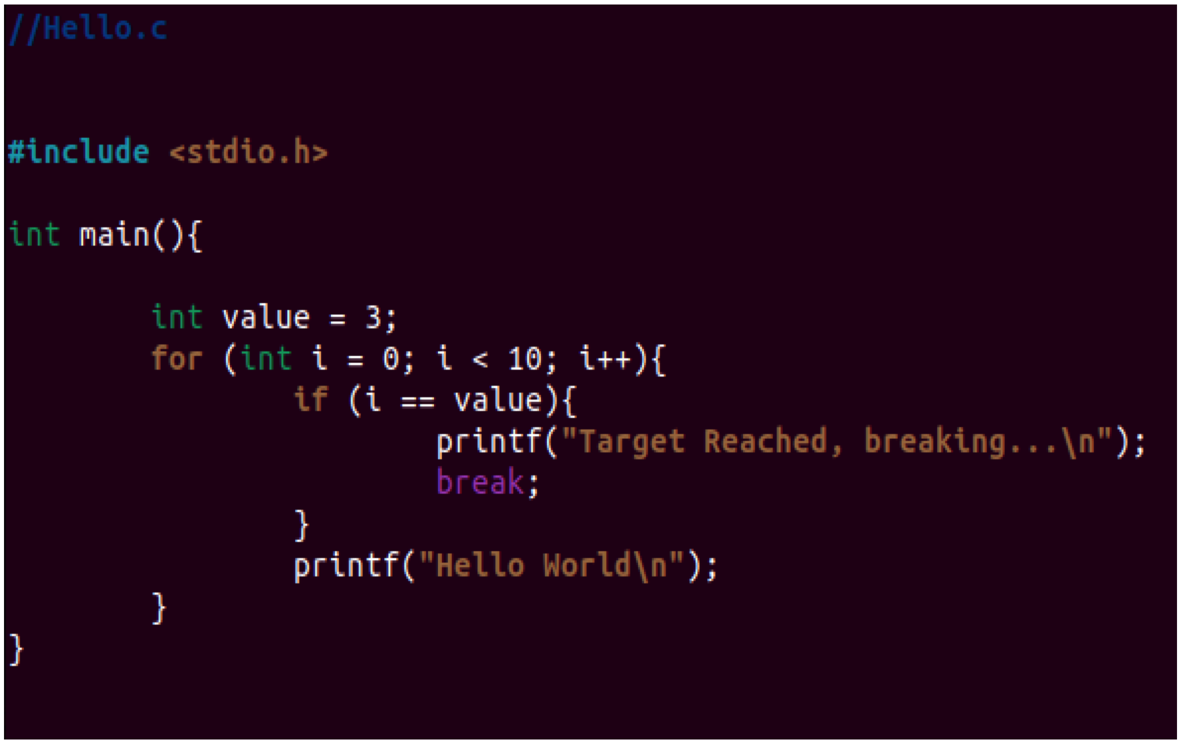
GNU Debugger, also called "gdb" is a command line debugger primarily used in UNIX systems to debug programs of many languages like C, C++, etc.

It was written by Richard Stallman in 1986 as a part of this GNU System, and since then it has been widely used as a goto debugger till today.

It provides a lot of features to help debug a program, which includes monitoring and modifying the values of internal variables at run time, calling functions independent of the actual program behavior, and much more.

❖ Explore the GDB commands for the following purpose

1) Running a program :



```
//Hello.c


#include <stdio.h>

int main(){

        int value = 3;
        for (int i = 0; i < 10; i++){
                if (i == value){
                        printf("Target Reached, breaking...\n");
                        break;
                }
                printf("Hello World\n");
        }
}
```

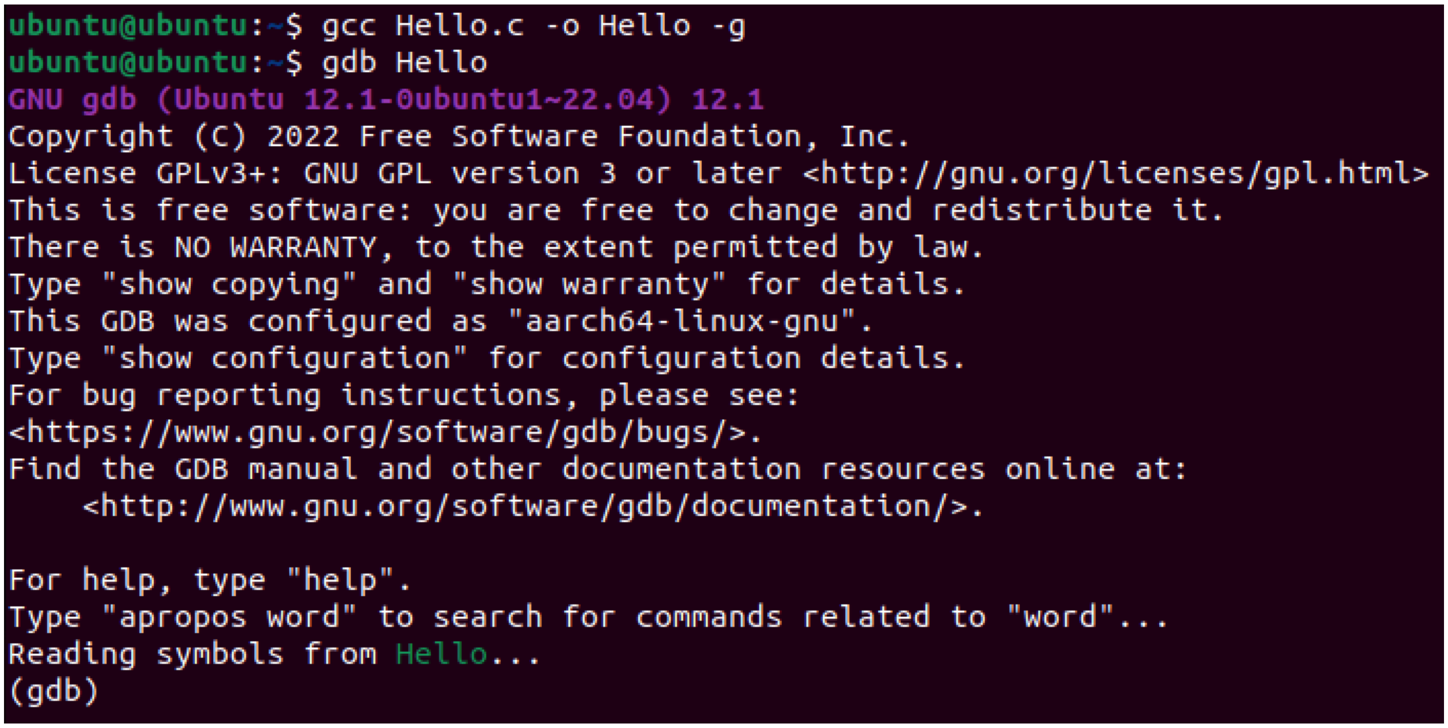
Generally, we do "gcc hello.c -o hello" to get the output.

But to generate a GDB accessible program, we will have to use the "-g" flag.

- "-g" flag is used to signal gcc to also generate a symbol table for gdb to read.

So "gcc hello.c -o hello -g".

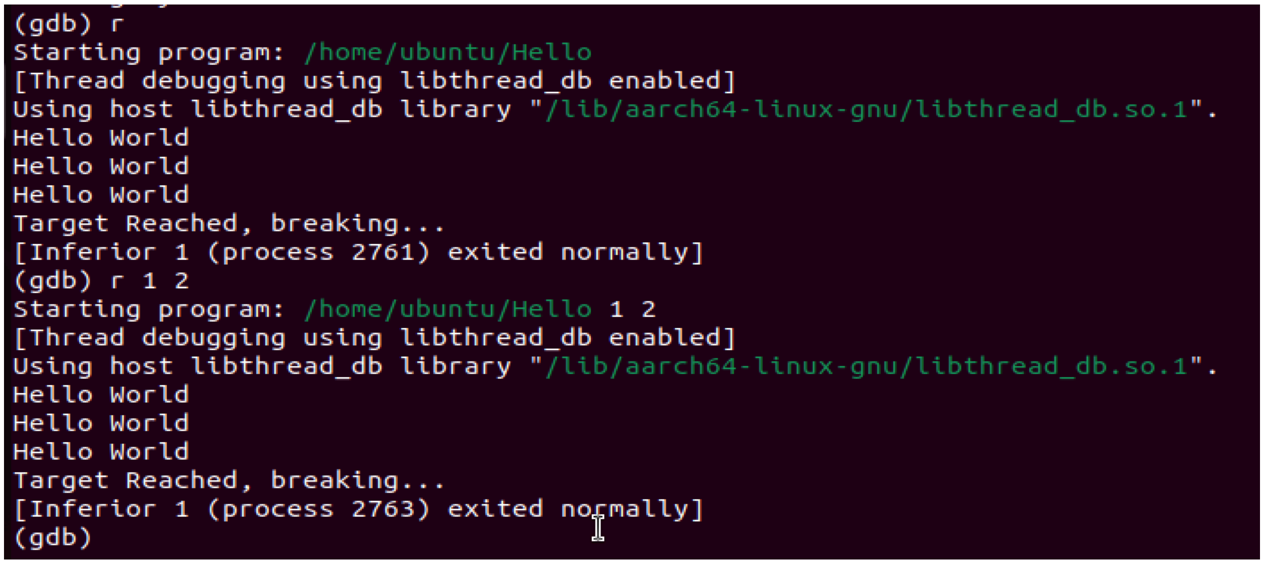
Now we just have to do "gdb hello" to start the debugger



```
ubuntu@ubuntu:~$ gcc Hello.c -o Hello -g
ubuntu@ubuntu:~$ gdb Hello
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 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 "aarch64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://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"...
Reading symbols from Hello...
(gdb)
```

Now we are inside gdb, to just run the program, we can do the following:

a. "r" to run the program
b. "r 1 2" to pass command-line arguments to the program
c. "r < file1" to feed a file for writing the outputs



```
(gdb) r
Starting program: /home/ubuntu/Hello
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".
Hello World
Hello World
Hello World
Target Reached, breaking...
[Inferior 1 (process 2761) exited normally]
(gdb) r 1 2
Starting program: /home/ubuntu/Hello 1 2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".
Hello World
Hello World
Hello World
Target Reached, breaking...
[Inferior 1 (process 2763) exited normally]
(gdb)
```

2) Loading Symbol Table:

Symbol table stores names of variables, functions and types defined within our program.

3) Setting a break-point:

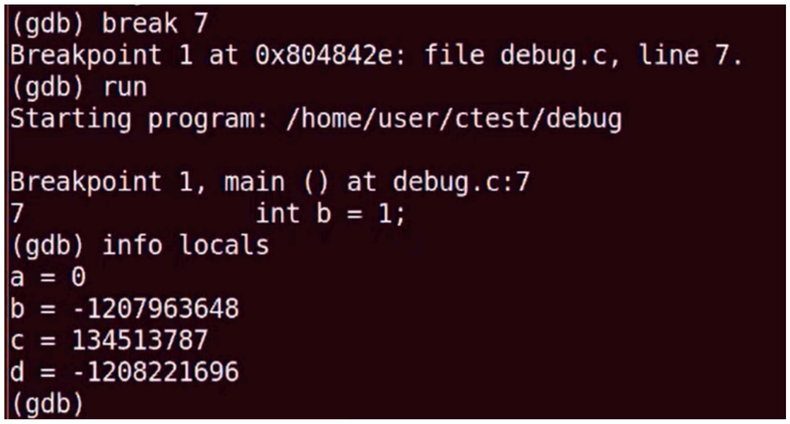
- There are many ways to set up break-points in the GNU GDB. Two majorly used ones include:

  a. Setting a break-point at the start of a function():

     break /*{function name}*/

  b. Setting a break-point at a specific line:

     break /*{line number}*/



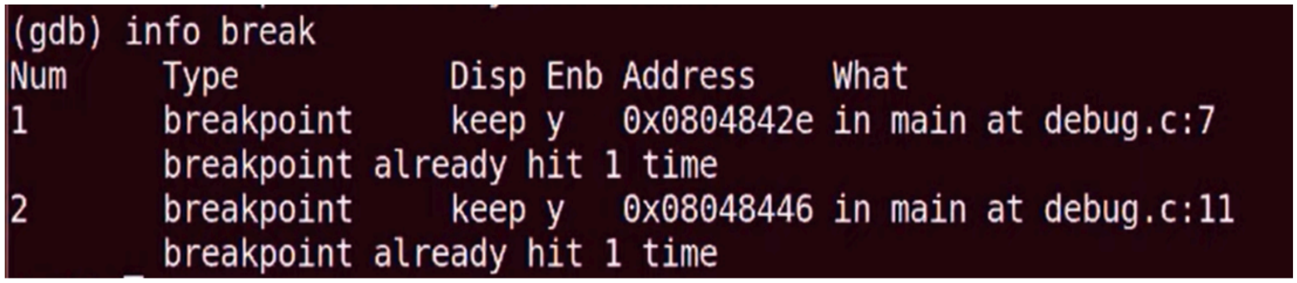
```
(gdb) break 7
Breakpoint 1 at 0x804842e: file debug.c, line 7.
(gdb) run
Starting program: /home/user/ctest/debug

Breakpoint 1, main () at debug.c:7
7               int b = 1;
(gdb) info locals
a = 0
b = -1207963648
c = 134513787
d = -1208221696
(gdb)
```

This command will add a break-point at the line number we specify. As soon as the execution reaches our line number, it will halt the execution process.

- After a Break Point is reached, we can view the state of every variable, function and the call stack up until that point.

- The command "info locals" will show us the state of the local variables at out break-point.

- The command “info break” will show us all the break-points currently held within our program.



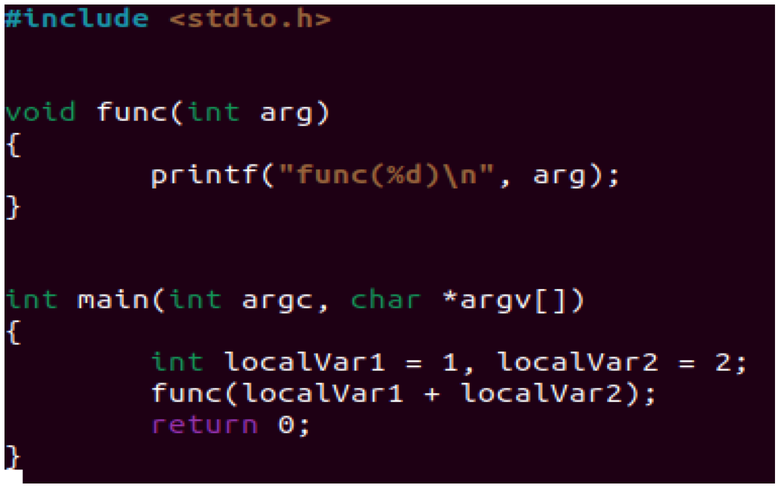
```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804842e in main at debug.c:7
        breakpoint already hit 1 time
2       breakpoint     keep y   0x08048446 in main at debug.c:11
        breakpoint already hit 1 time
```

- The command “continue” will resume the execution which was previously halted after reaching a break-point.
- The command “delete /*{breakpoint number}*/ will delete a break-point.

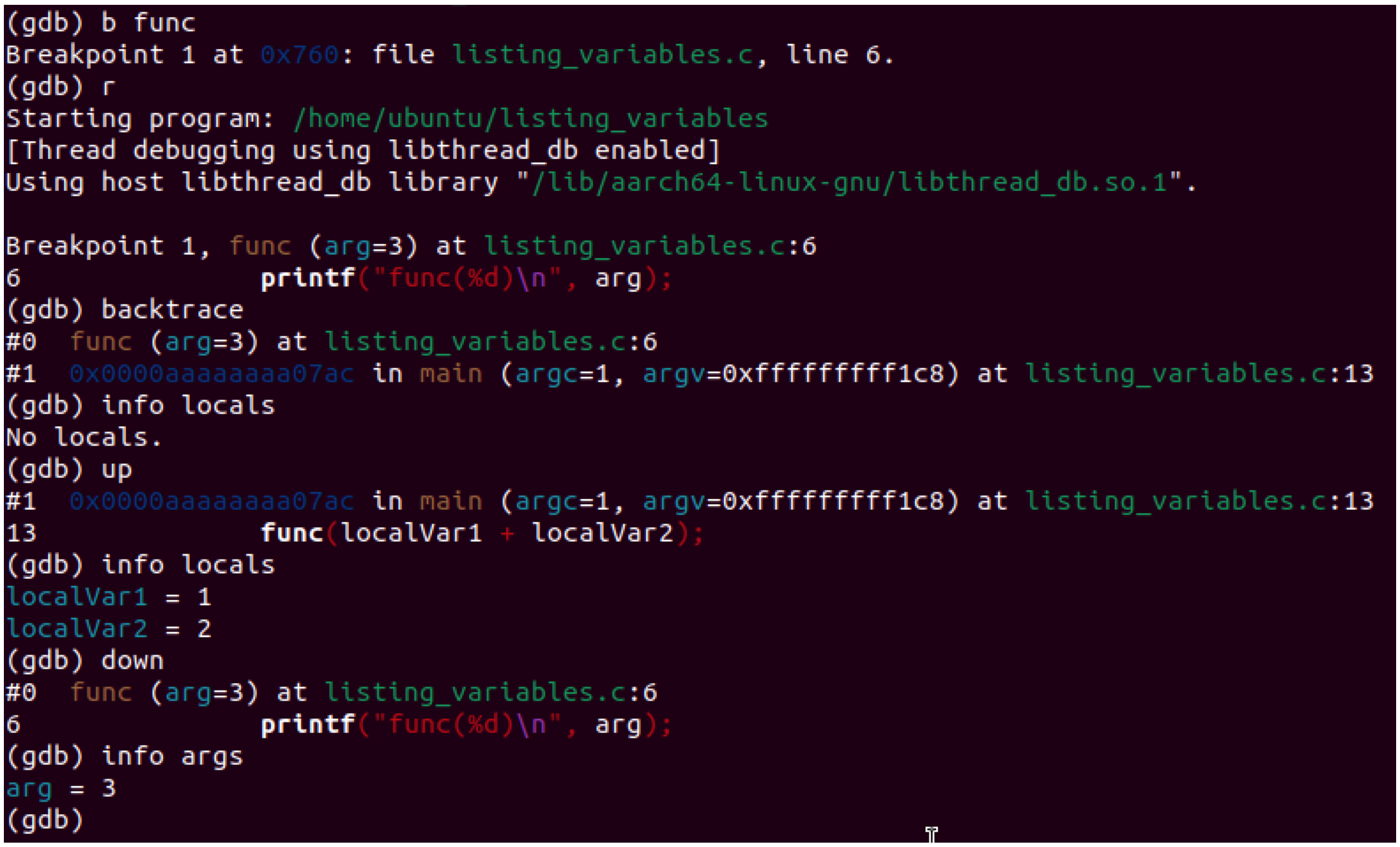
4) Listing variables and examining their values:

- The info locals command displays the local variable values in the current frame. You can select frames using the frame, up and down.
- Note that the info locals command does not display the information about the function arguments.
- Use the info args command to list function arguments.

We will run the program, set a breakpoint in func() and use the info locals command to display the local variables in main():



```
#include <stdio.h>

void func(int arg)
{
        printf("func(%d)\n", arg);
}

int main(int argc, char *argv[])
{
        int localVar1 = 1, localVar2 = 2;
        func(localVar1 + localVar2);
        return 0;
}
```



```
(gdb) b func
Breakpoint 1 at 0x760: file listing_variables.c, line 6.
(gdb) r
Starting program: /home/ubuntu/listing_variables
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".

Breakpoint 1, func (arg=3) at listing_variables.c:6
6                printf("func(%d)\n", arg);
(gdb) backtrace
#0  func (arg=3) at listing_variables.c:6
#1  0x0000aaaaaaaa07ac in main (argc=1, argv=0xffffffff1c8) at listing_variables.c:13
(gdb) info locals
No locals.
(gdb) up
#1  0x0000aaaaaaaa07ac in main (argc=1, argv=0xffffffff1c8) at listing_variables.c:13
13               func(localVar1 + localVar2);
(gdb) info locals
localVar1 = 1
localVar2 = 2
(gdb) down
#0  func (arg=3) at listing_variables.c:6
6                printf("func(%d)\n", arg);
(gdb) info args
arg = 3
(gdb)
```

5) Printing content of an array or contiguous memory:

- **Syntax:**

  set print array on

  set print array off

  show print array

- **Modes:**

  ON: GDB will display the values of arrays in a simple one-line format (e.g. $1 = {1, 2, 3}).

  OFF: GDB will display the values of arrays using a longer multi-line format.

- **Default mode:**

  The default value for the print array setting is 'off'.

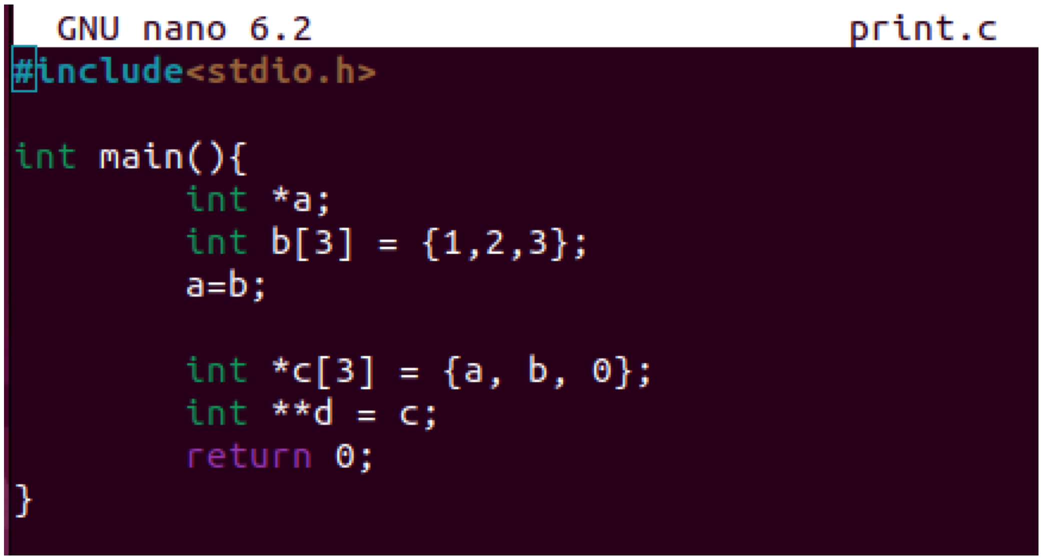
- **Remarks:**

The set print array command can be used together with the set print array-indexes command to further customize the output of the array contents.

Below is a log of sample GDB session illustrating how set print array command affects the output of the print command:

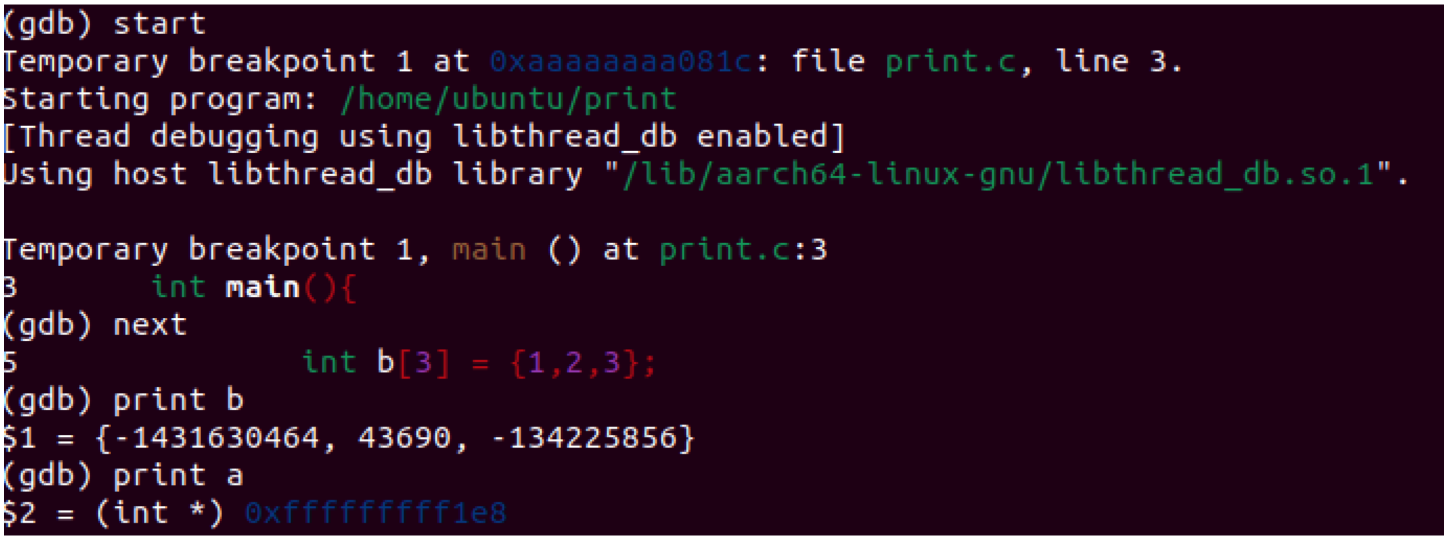
```
(gdb) start
Temporary breakpoint 1 at 0x80483f3: file test.cpp, line 5.
Starting program: /home/bazis/test

Temporary breakpoint 1, main (argc=1, argv=0xbffff064) at test.cpp:5
5 int testArray[] = {1, 2, 3};
(gdb) next
6 return 0;
(gdb) print testArray
$1 = {1, 2, 3}
(gdb) show print array
Prettyprinting of arrays is off.
(gdb) set print array on
(gdb) print testArray
$2 = {1,
2,
3}
```

Example:



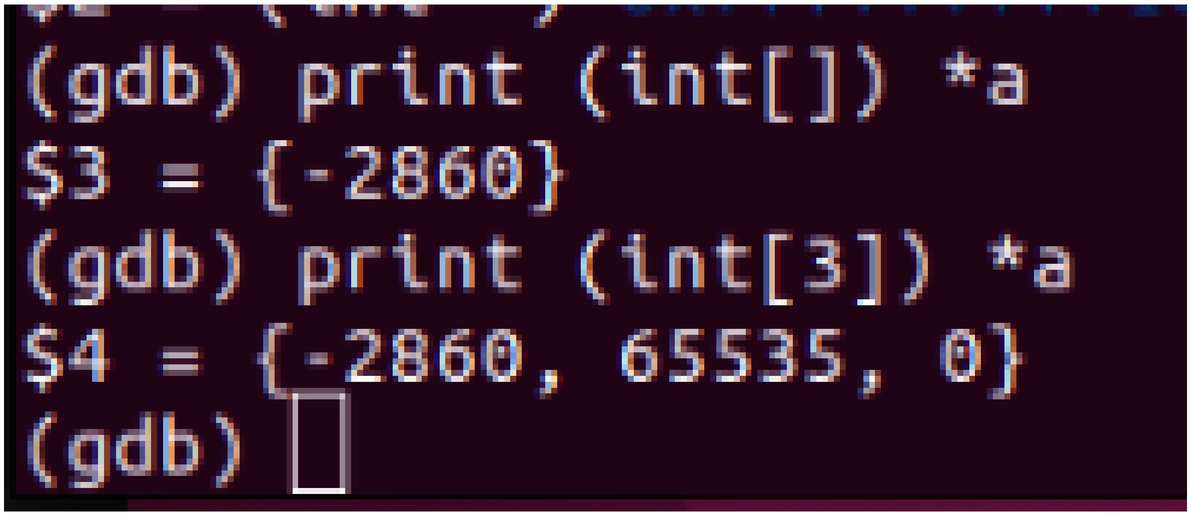
```
  GNU nano 6.2                    print.c
#include<stdio.h>

int main(){
        int *a;
        int b[3] = {1,2,3};
        a=b;

        int *c[3] = {a, b, 0};
        int **d = c;
        return 0;
}
```

While debugging the above code if we do:



```
(gdb) start
Temporary breakpoint 1 at 0xaaaaaaaaa081c: file print.c, line 3.
Starting program: /home/ubuntu/print
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 1, main () at print.c:3
3       int main(){
(gdb) next
5               int b[3] = {1,2,3};
(gdb) print b
$1 = {-1431630464, 43690, -134225856}
(gdb) print a
$2 = (int *) 0xfffffffff1e8
```

Both works but in order to print a as an array:



```
(gdb) print (int[]) *a
$3 = {-2860}
(gdb) print (int[3]) *a
$4 = {-2860, 65535, 0}
(gdb) 
```

and when we specify the size it gets better.

6) Printing function arguments:

Displays information about the function arguments corresponding to the current stack frame.

Syntax

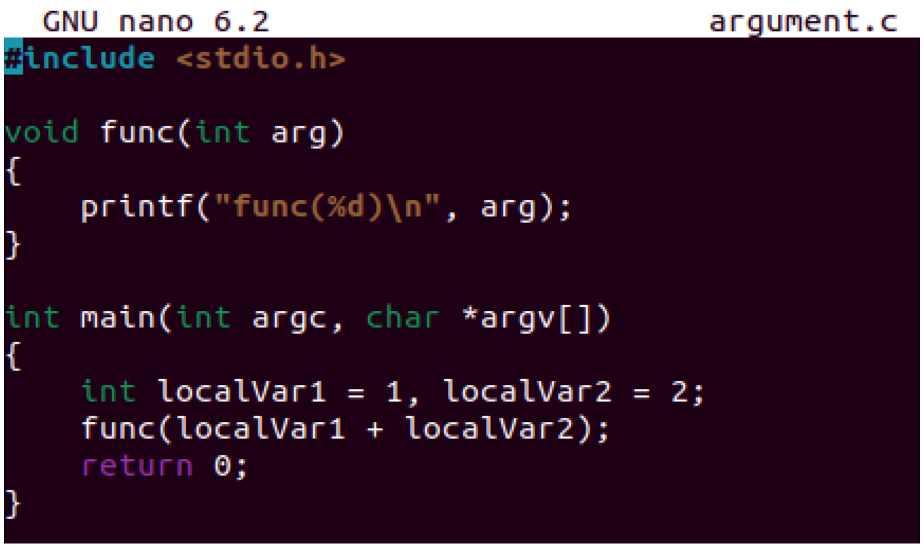
info args

Remarks

- The info args command displays the function argument values of the current frame. You can select frames using the frame, up and down commands.
- Note that the info args command does not display the information about the local variables. Use the info locals command to list local variables.
- Do not confuse the info args with the set args command. The set args command sets the command-line arguments for the debugged program, while the info args displays the arguments for the current function. The arguments provided via set args will be available via argv in the main() function.

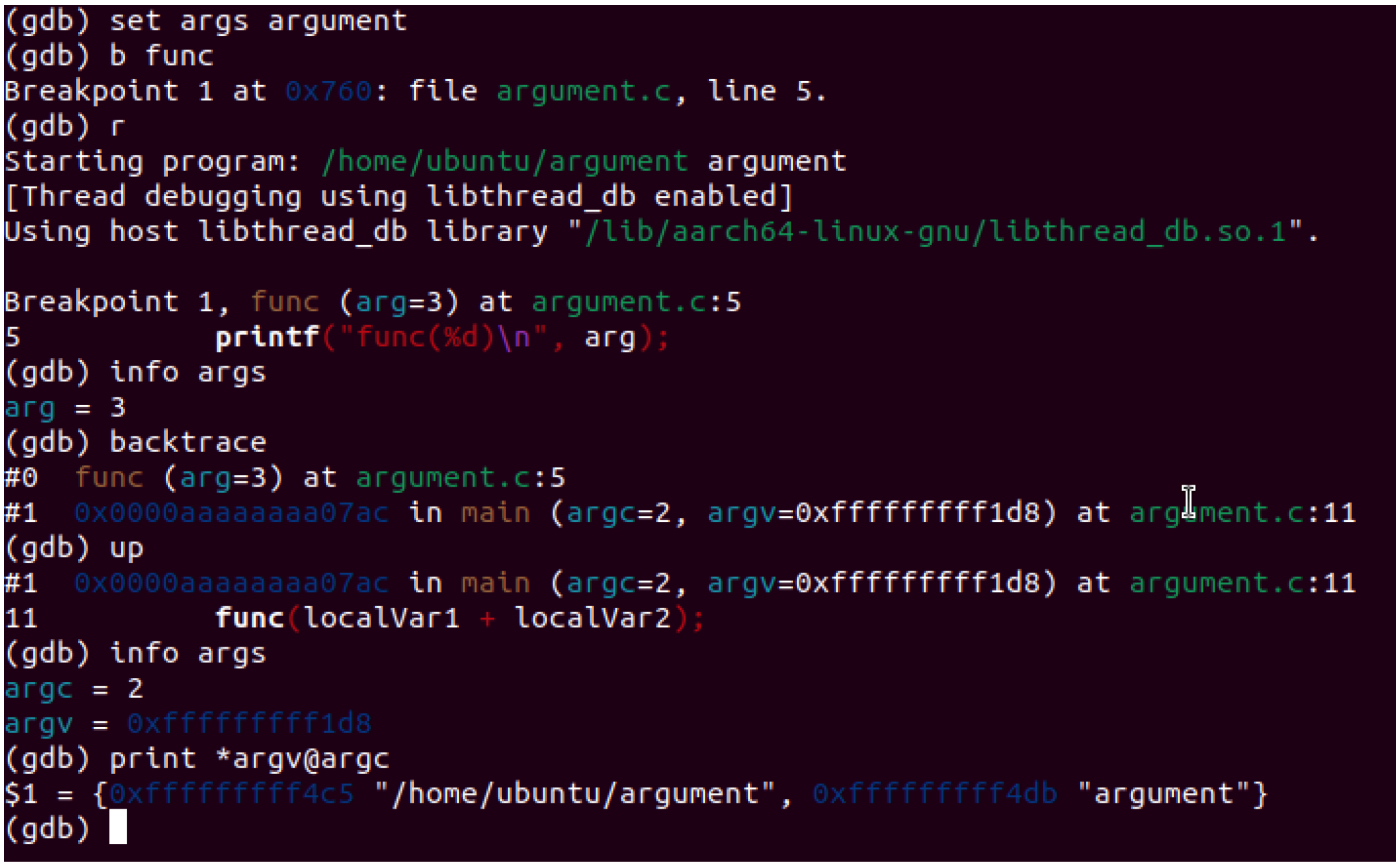
Examples

To demonstrate the info args command we will debug a sample program listed below:



```
  GNU nano 6.2                              argument.c
#include <stdio.h>

void func(int arg)
{
    printf("func(%d)\n", arg);
}

int main(int argc, char *argv[])
{
    int localVar1 = 1, localVar2 = 2;
    func(localVar1 + localVar2);
    return 0;
}
```

We will run the program, set a breakpoint in func() and use the info args command to display the argument values:



```
(gdb) set args argument
(gdb) b func
Breakpoint 1 at 0x760: file argument.c, line 5.
(gdb) r
Starting program: /home/ubuntu/argument argument
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".

Breakpoint 1, func (arg=3) at argument.c:5
5           printf("func(%d)\n", arg);
(gdb) info args
arg = 3
(gdb) backtrace
#0  func (arg=3) at argument.c:5
#1  0x0000aaaaaaaa07ac in main (argc=2, argv=0xfffffffff1d8) at argument.c:11
(gdb) up
#1  0x0000aaaaaaaa07ac in main (argc=2, argv=0xfffffffff1d8) at argument.c:11
11          func(localVar1 + localVar2);
(gdb) info args
argc = 2
argv = 0xfffffffff1d8
(gdb) print *argv@argc
$1 = {0xfffffffff4c5 "/home/ubuntu/argument", 0xfffffffff4db "argument"}
(gdb) 
```

7) Next, Continue, Set command:

Next - Is used to step through code

- **If the instruction is a single line of code it displays it.**

- **If the line is a function call, it executes the entire function call in one step.**

There is a variation of the call next which is nexti (ni). According to the help pages of gdb below is it's functionality as compared to next (n)

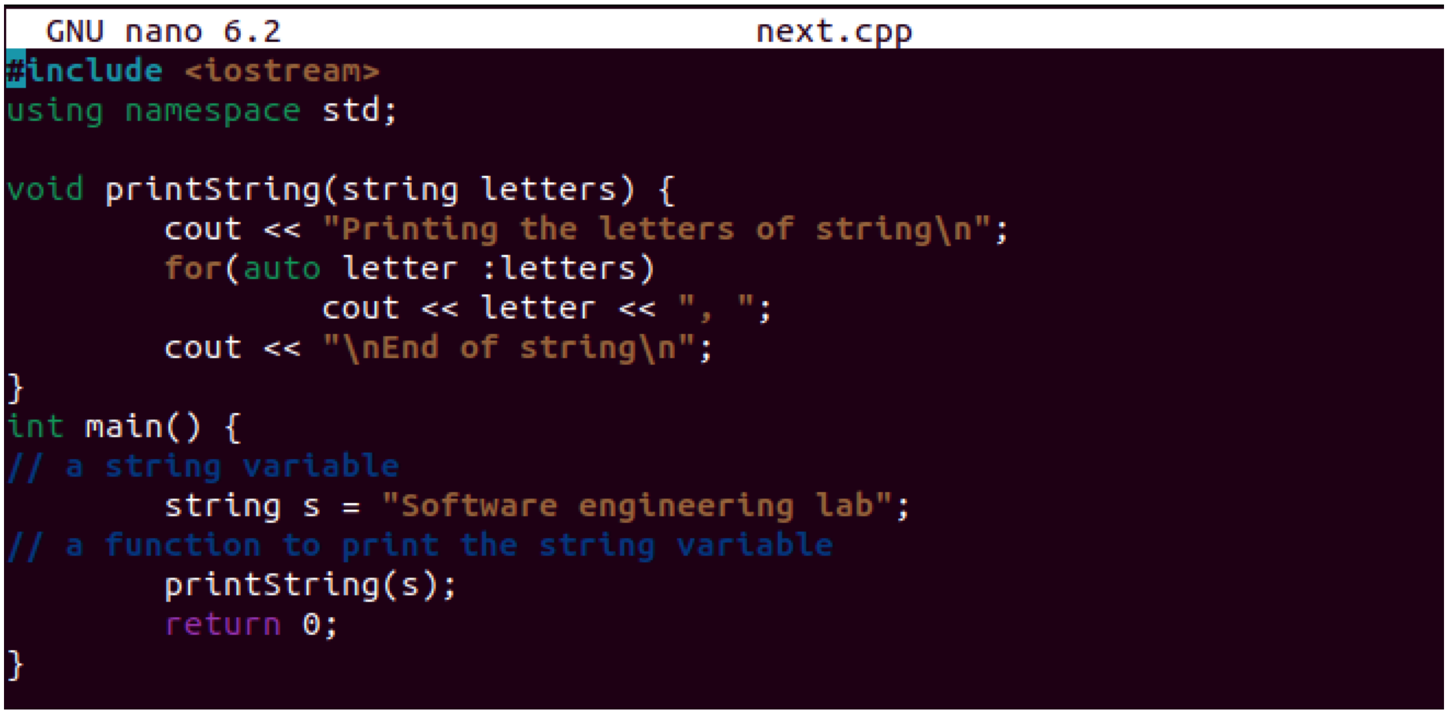
next

- **Step program, proceeding through subroutine calls.**
- **Usage: next [N]**
- **Unlike "step", if the current source line calls a subroutine, this command does not enter the subroutine, but instead steps over the call, in effect treating it as a single source line.**

nexti

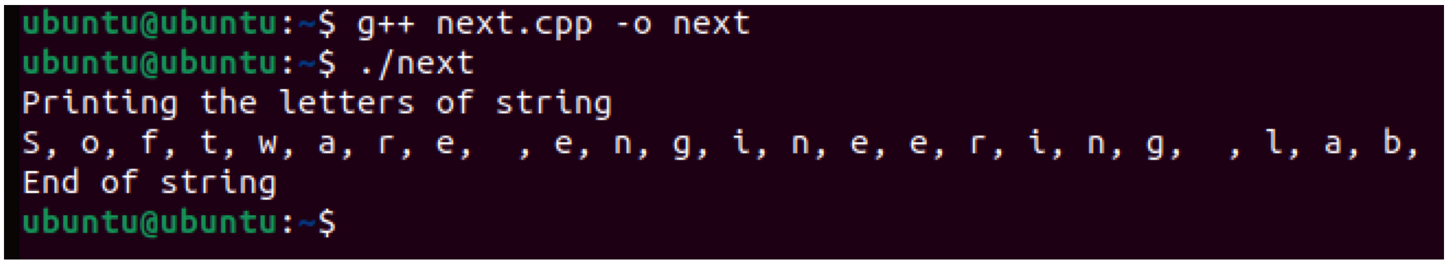
- **Step one instruction, but proceed through subroutine calls.**
- **Usage: nexti [N]**
- **Argument N means step N times (or till the program stops for another reason).**

Below is an example demonstrating the use of next:



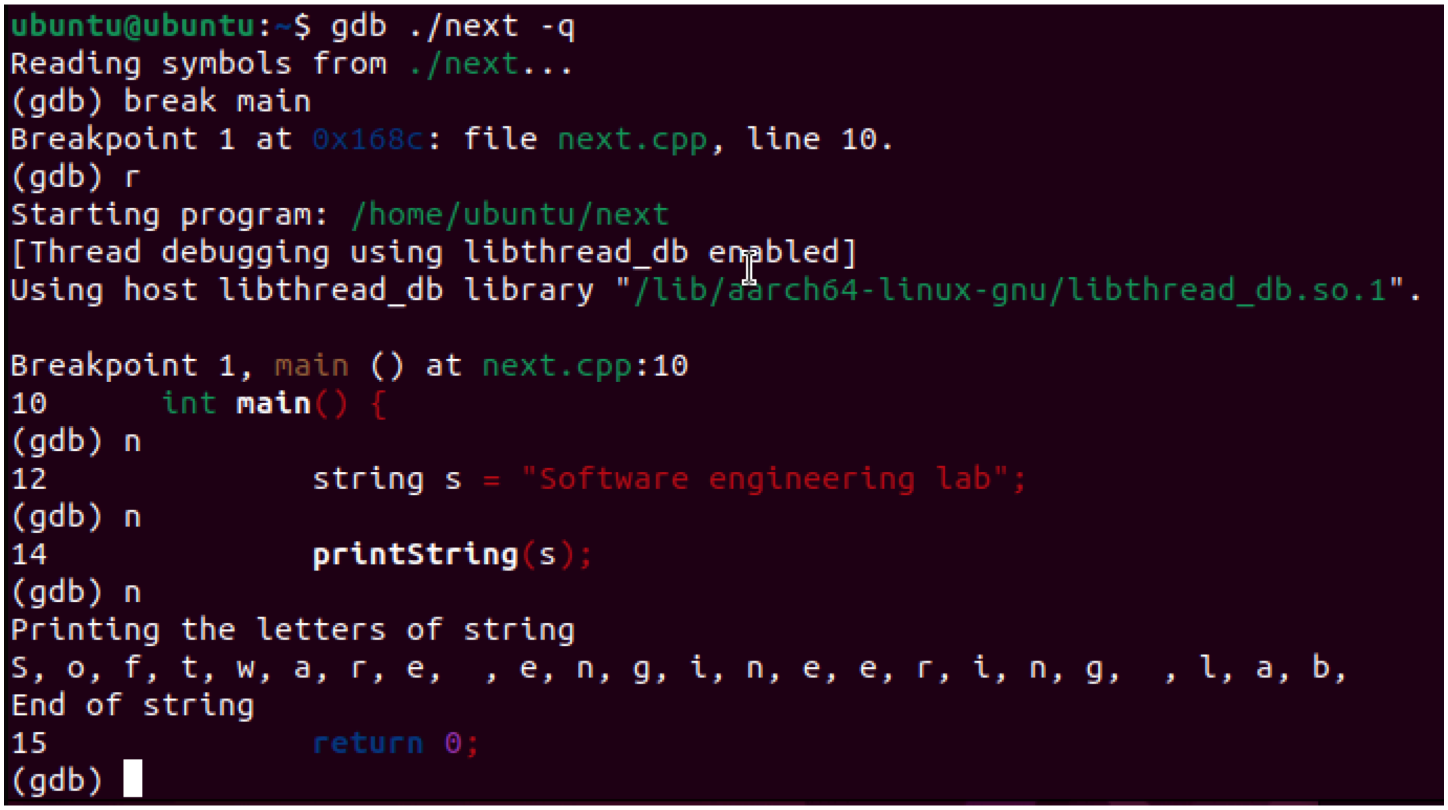
```
  GNU nano 6.2                                next.cpp
#include <iostream>
using namespace std;

void printString(string letters) {
        cout << "Printing the letters of string\n";
        for(auto letter :letters)
                cout << letter << ", ";
        cout << "\nEnd of string\n";
}
int main() {
// a string variable
        string s = "Software engineering lab";
// a function to print the string variable
        printString(s);
        return 0;
}
```

The output of this code is:



```
ubuntu@ubuntu:~$ g++ next.cpp -o next
ubuntu@ubuntu:~$ ./next
Printing the letters of string
S, o, f, t, w, a, r, e,  , e, n, g, i, n, e, e, r, i, n, g,  , l, a, b,
End of string
ubuntu@ubuntu:~$
```

For demonstration purposes, a break-point has been added at main and the program is then run.



```
ubuntu@ubuntu:~$ gdb ./next -q
Reading symbols from ./next...
(gdb) break main
Breakpoint 1 at 0x168c: file next.cpp, line 10.
(gdb) r
Starting program: /home/ubuntu/next
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at next.cpp:10
10      int main() {
(gdb) n
12              string s = "Software engineering lab";
(gdb) n
14              printString(s);
(gdb) n
Printing the letters of string
S, o, f, t, w, a, r, e,  , e, n, g, i, n, e, e, r, i, n, g,  , l, a, b,
End of string
15              return 0;
(gdb) 
```

After adding a break-point at main, we run and step through the code. It can be seen that when the code happens to be a single line, next simple displays the code. However when it hits the function call printString(s), it executes the entire function rather than stepping through it. The difference will be more visible after the demonstration of step 8.

Continue - is used to resume normal execution of a program until it ends, crashes or a break-point is encountered.

- **Continues program execution after a breakpoint.**
- Syntax
  - ➢ continue
  - ➢ continue [*Repeat count*]
  - ➢ c
  - ➢ c *[Repeat count]*

- **Parameters**

  Repeat count

  If this parameter is specified, GDB will auto-continue the next *Repeat count* - 1 times when the current breakpoint is hit.

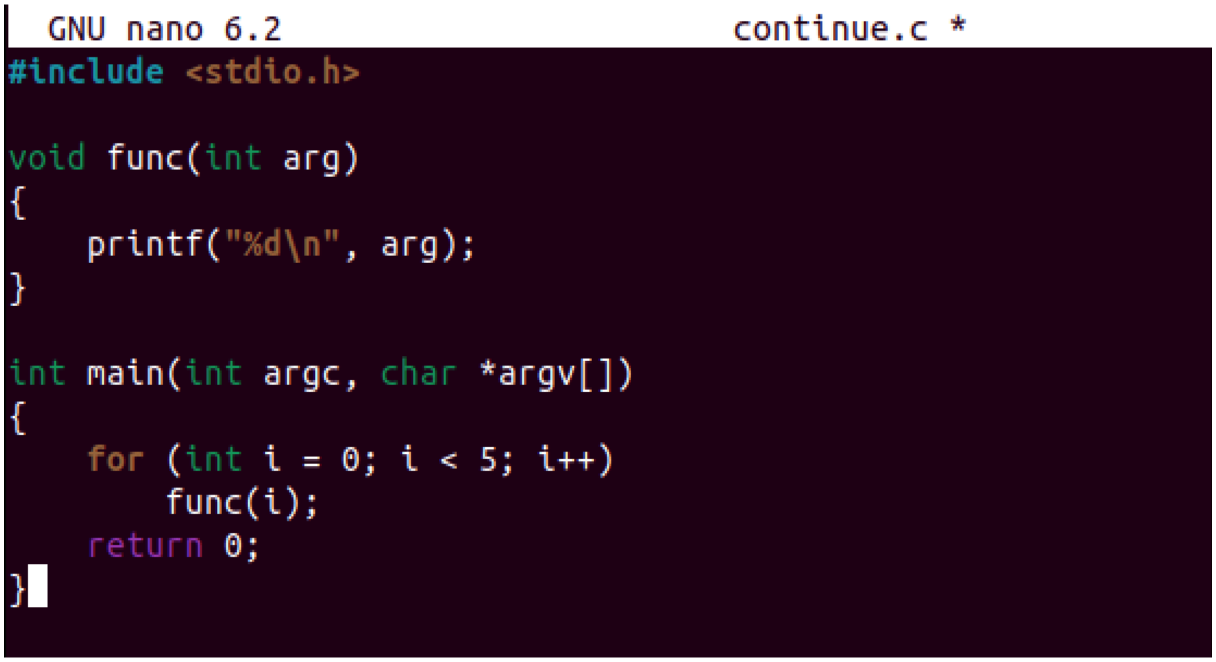
- **Remarks**

The continue is also used to start debugging in the following cases:

- ❖ To resume a process after attaching to it with attach
- ❖ To start debugging with gdbserver

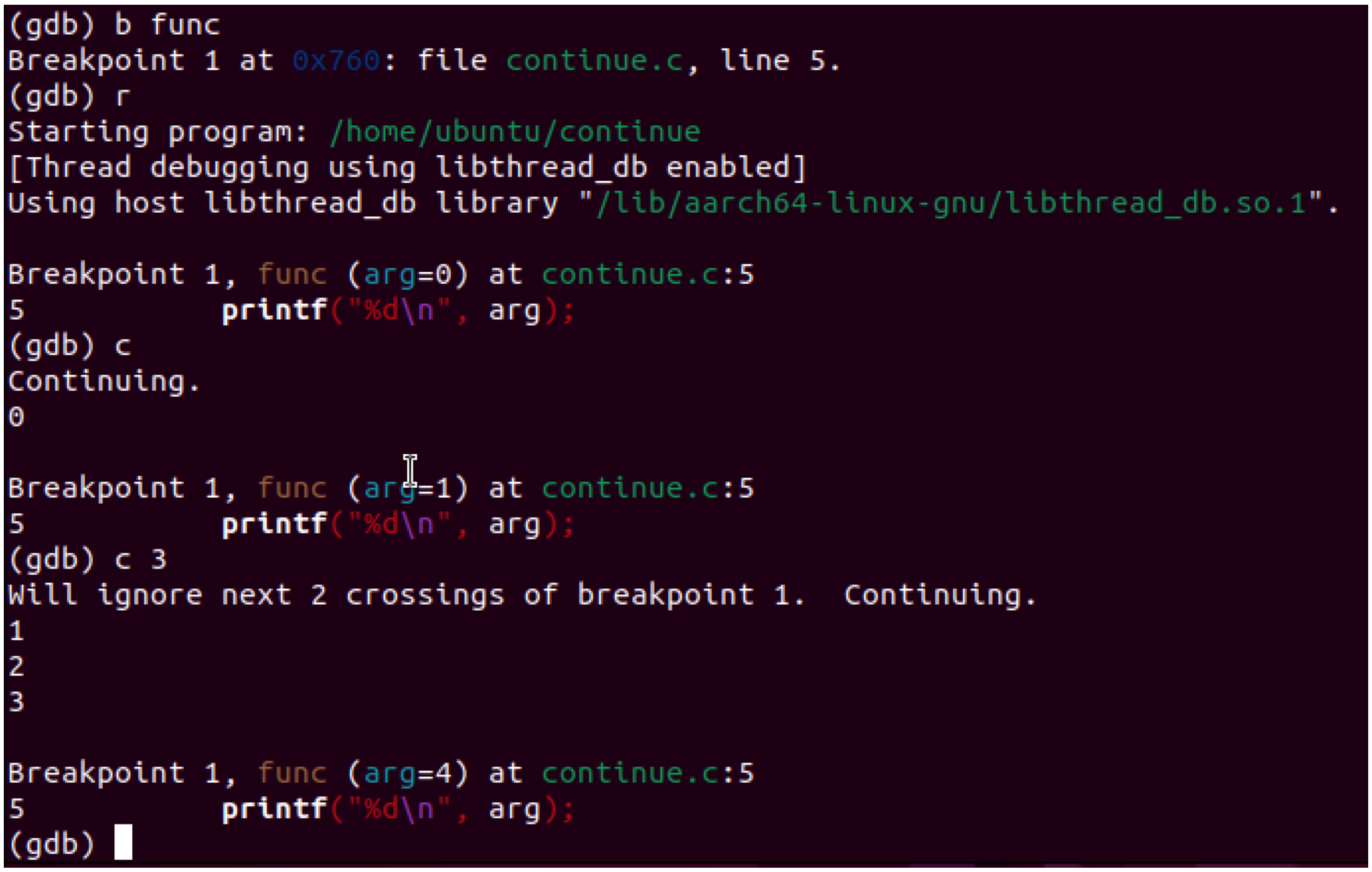
- **Examples**

This example illustrates the use of the *Repeat count* parameter. The following program is being debugged:



```
  GNU nano 6.2                             continue.c *
#include <stdio.h>

void func(int arg)
{
    printf("%d\n", arg);
}

int main(int argc, char *argv[])
{
    for (int i = 0; i < 5; i++)
        func(i);
    return 0;
}
```

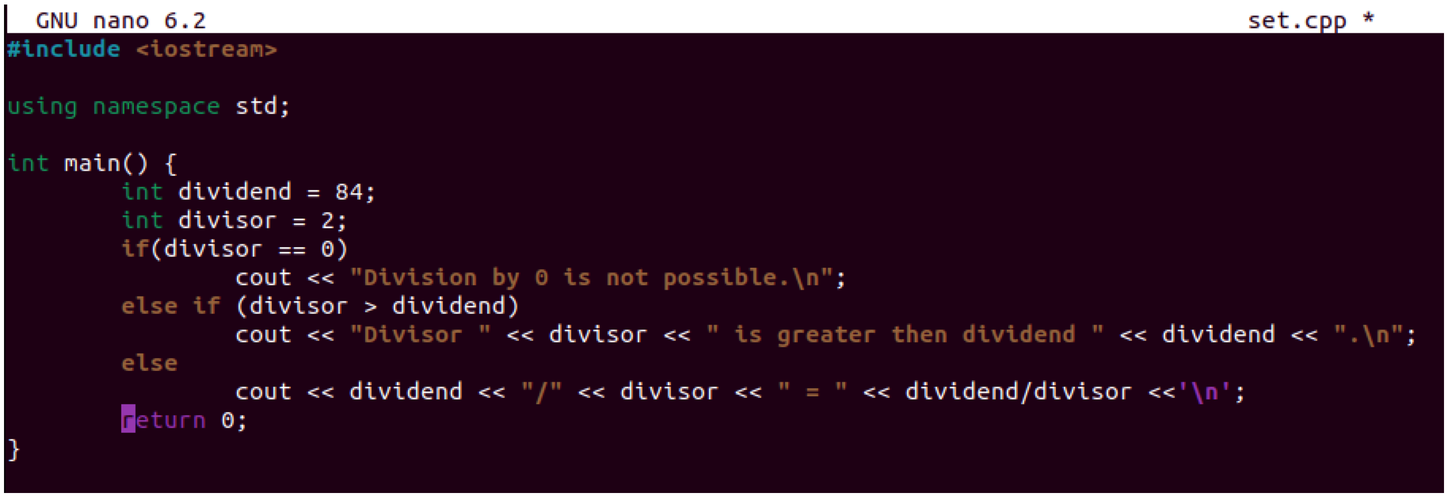
When a continue command is issued without any parameters, GDB breaks in the next loop iteration. When a repeat count of 3 is specified, GDB skips the next 2 iterations. In case of a single breakpoint this is equivalent to issuing the continue command 3 times.



```
(gdb) b func
Breakpoint 1 at 0x760: file continue.c, line 5.
(gdb) r
Starting program: /home/ubuntu/continue
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".

Breakpoint 1, func (arg=0) at continue.c:5
5           printf("%d\n", arg);
(gdb) c
Continuing.
0

Breakpoint 1, func (arg=1) at continue.c:5
5           printf("%d\n", arg);
(gdb) c 3
Will ignore next 2 crossings of breakpoint 1.  Continuing.
1
2
3

Breakpoint 1, func (arg=4) at continue.c:5
5           printf("%d\n", arg);
(gdb) 
```

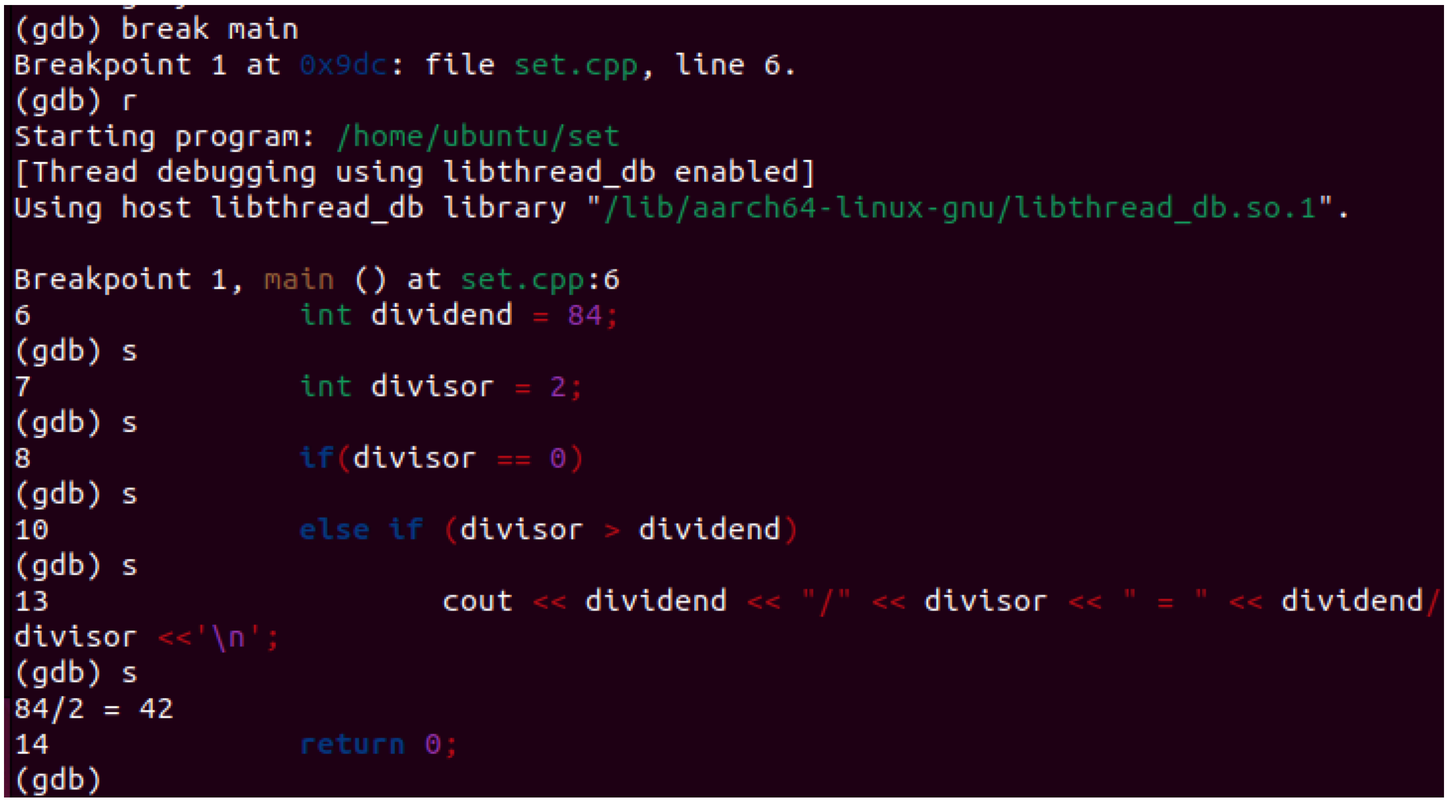
Set - is used to set the value of a variable in the program during runtime

The value of the variable is not changed in the code but is rather used only for evaluation purposes during runtime.



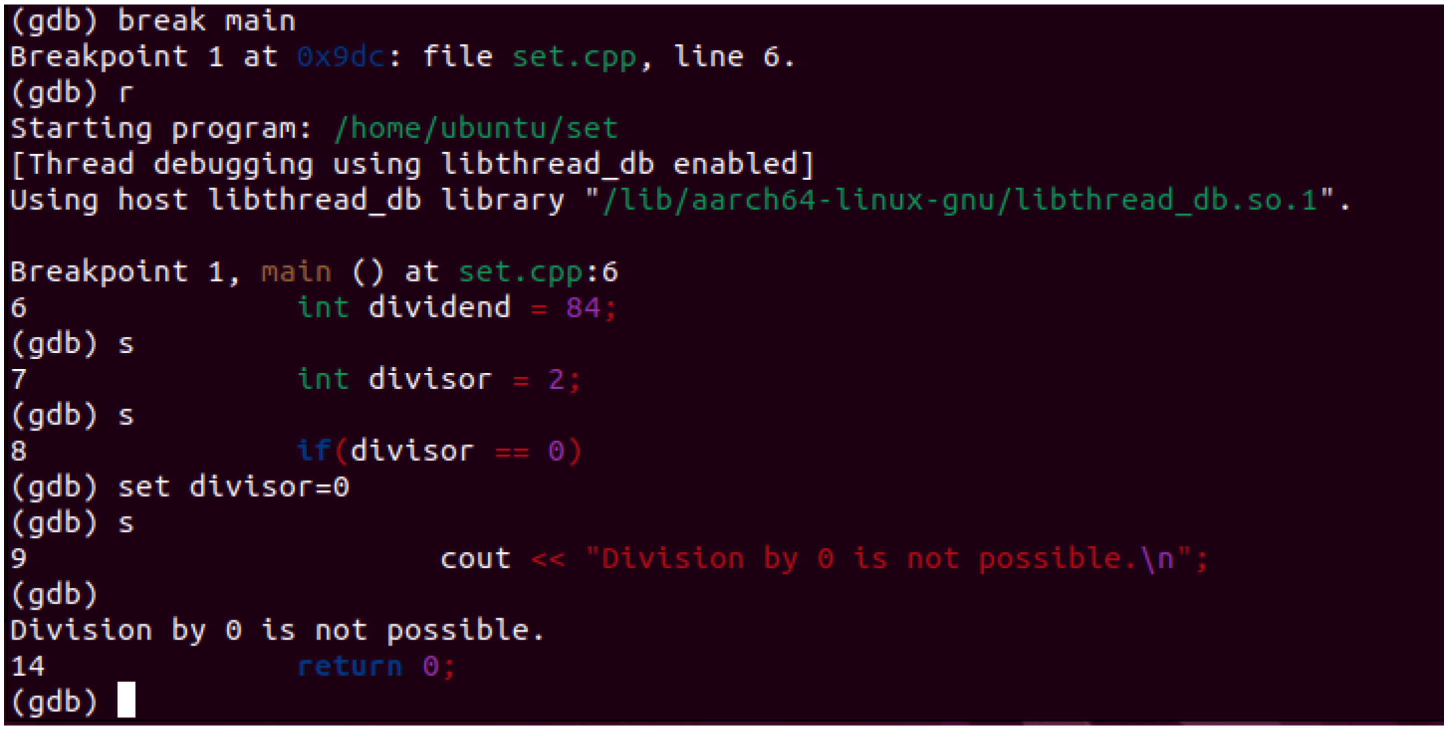
```
  GNU nano 6.2                                                            set.cpp *
#include <iostream>

using namespace std;

int main() {
        int dividend = 84;
        int divisor = 2;
        if(divisor == 0)
                cout << "Division by 0 is not possible.\n";
        else if (divisor > dividend)
                cout << "Divisor " << divisor << " is greater then dividend " << dividend << ".\n";
        else
                cout << dividend << "/" << divisor << " = " << dividend/divisor <<'\n';
        return 0;
}
```

Using set to set values for divisor to check resilience of code.



```
(gdb) break main
Breakpoint 1 at 0x9dc: file set.cpp, line 6.
(gdb) r
Starting program: /home/ubuntu/set
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at set.cpp:6
6               int dividend = 84;
(gdb) s
7               int divisor = 2;
(gdb) s
8               if(divisor == 0)
(gdb) s
10              else if (divisor > dividend)
(gdb) s
13                      cout << dividend << "/" << divisor << " = " << dividend/
divisor <<'\n';
(gdb) s
84/2 = 42
14              return 0;
(gdb)
```

The condition divisor > dividend was evaluated at run-time by changing the runtime value of the variable divisor.



```
(gdb) break main
Breakpoint 1 at 0x9dc: file set.cpp, line 6.
(gdb) r
Starting program: /home/ubuntu/set
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at set.cpp:6
6               int dividend = 84;
(gdb) s
7               int divisor = 2;
(gdb) s
8               if(divisor == 0)
(gdb) set divisor=0
(gdb) s
9                       cout << "Division by 0 is not possible.\n";
(gdb)
Division by 0 is not possible.
14              return 0;
(gdb) 
```

The condition divisor == 0 evaluated to true by setting the value of divisor variable at runtime.

8) Single stepping into function:

Single stepping into a function means to execute every line of the function body one by one rather than treating it as a single line and executing it's body in one go as done by next.

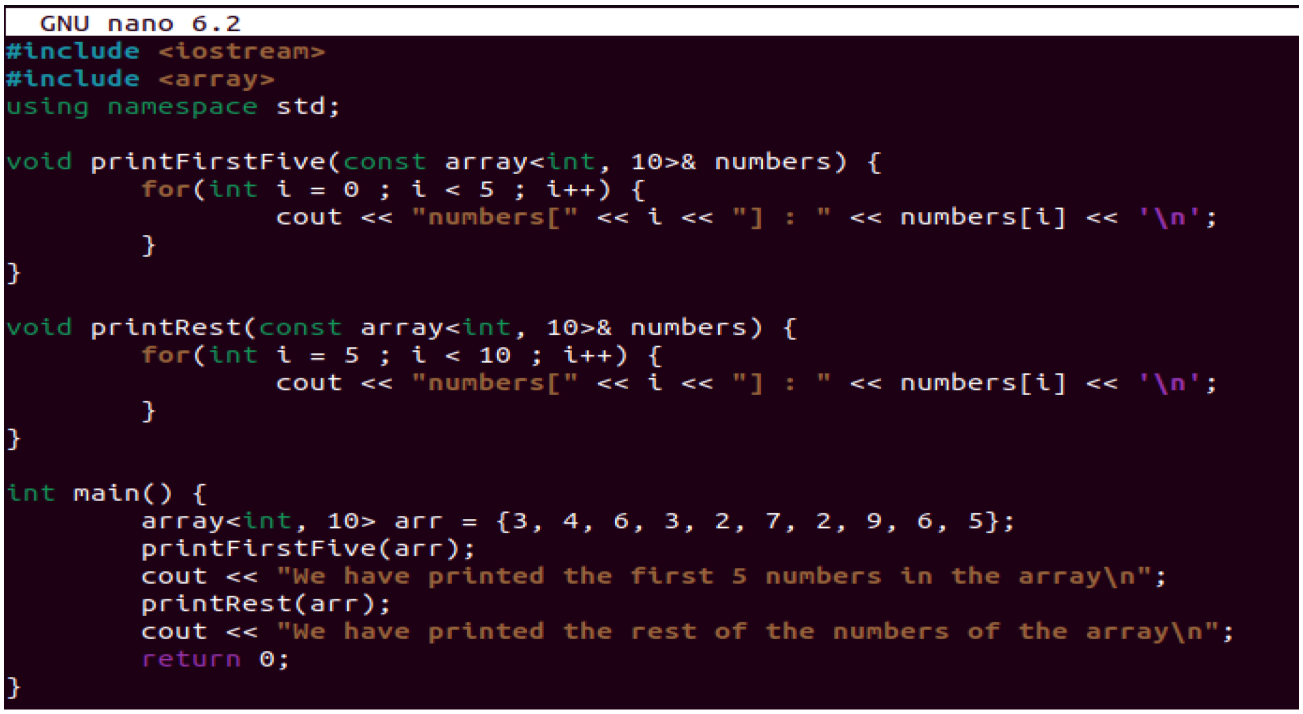
Variation stepi is also available. According to the help pages,

- **stepi (si)**
  - Step one instruction exactly.
  - Usage: stepi [N]
  - Argument N means step N times (or till the program stops for another reason).

- **step (s)**
  - Step program until it reaches a different source line.
  - Usage: step [N]
  - Argument N means step N times (or till the program stops for another reason).

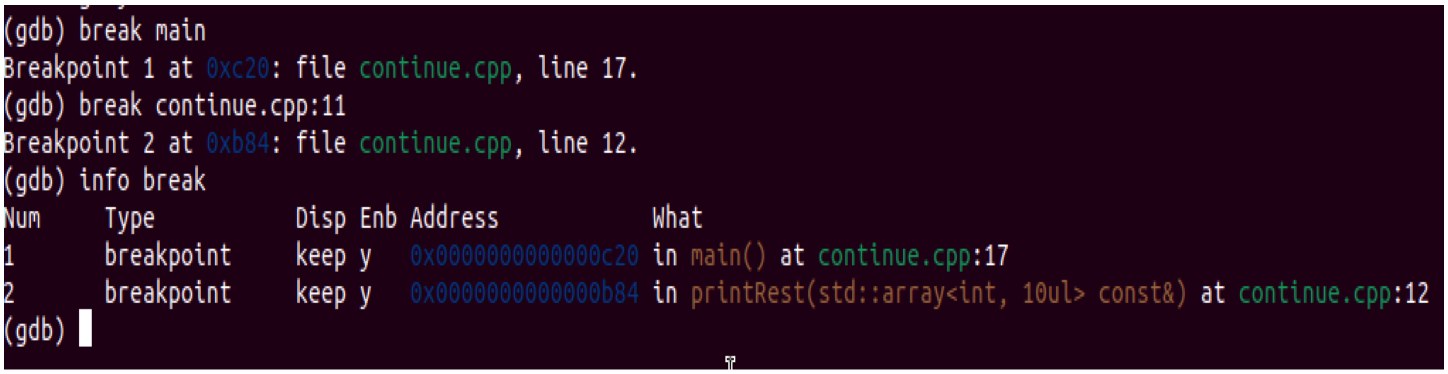
9) Listing all break-point:

info breaks is the command to list all break-points in the current file



```
  GNU nano 6.2
#include <iostream>
#include <array>
using namespace std;

void printFirstFive(const array<int, 10>& numbers) {
        for(int i = 0 ; i < 5 ; i++) {
                cout << "numbers[" << i << "] : " << numbers[i] << '\n';
        }
}

void printRest(const array<int, 10>& numbers) {
        for(int i = 5 ; i < 10 ; i++) {
                cout << "numbers[" << i << "] : " << numbers[i] << '\n';
        }
}

int main() {
        array<int, 10> arr = {3, 4, 6, 3, 2, 7, 2, 9, 6, 5};
        printFirstFive(arr);
        cout << "We have printed the first 5 numbers in the array\n";
        printRest(arr);
        cout << "We have printed the rest of the numbers of the array\n";
        return 0;
}
```

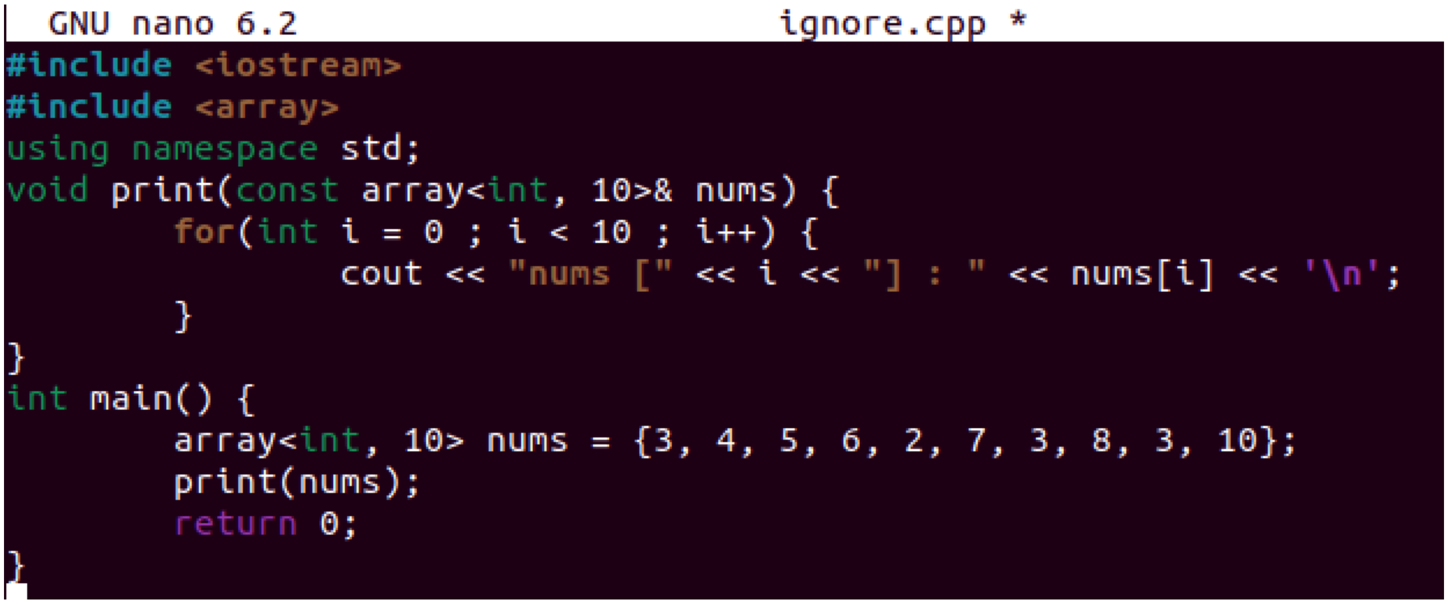
Using the above code, break-points are added and then displayed. Two different ways of adding a break are used; one through using functionName and another by using filename:lineNumber.



```
(gdb) break main
Breakpoint 1 at 0xc20: file continue.cpp, line 17.
(gdb) break continue.cpp:11
Breakpoint 2 at 0xb84: file continue.cpp, line 12.
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000000c20 in main() at continue.cpp:17
2       breakpoint     keep y   0x0000000000000b84 in printRest(std::array<int, 10ul> const&) at continue.cpp:12
(gdb) 
```

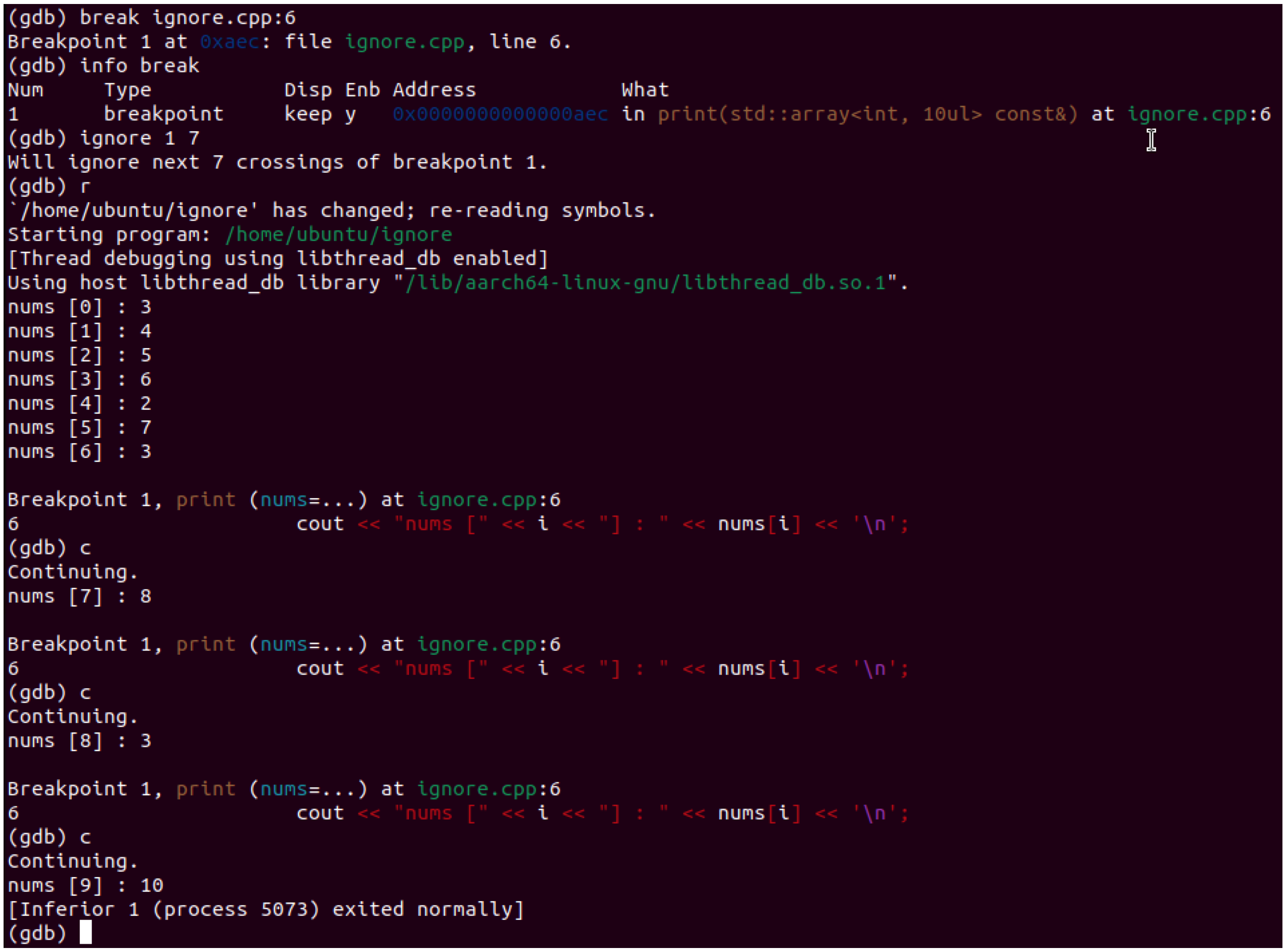
10) Ignoring a break-point for N occurrence:

ignore [breakpointnumber] [x] - ignores break-point number (this is done automatically by the system while setting a breakpoint) breakpointnumber until x hits are registered under it.



```
  GNU nano 6.2                                  ignore.cpp *
#include <iostream>
#include <array>
using namespace std;
void print(const array<int, 10>& nums) {
        for(int i = 0 ; i < 10 ; i++) {
                cout << "nums [" << i << "] : " << nums[i] << '\n';
        }
}
int main() {
        array<int, 10> nums = {3, 4, 5, 6, 2, 7, 3, 8, 3, 10};
        print(nums);
        return 0;
}
```

Setting a break-point at line 7 and running the code

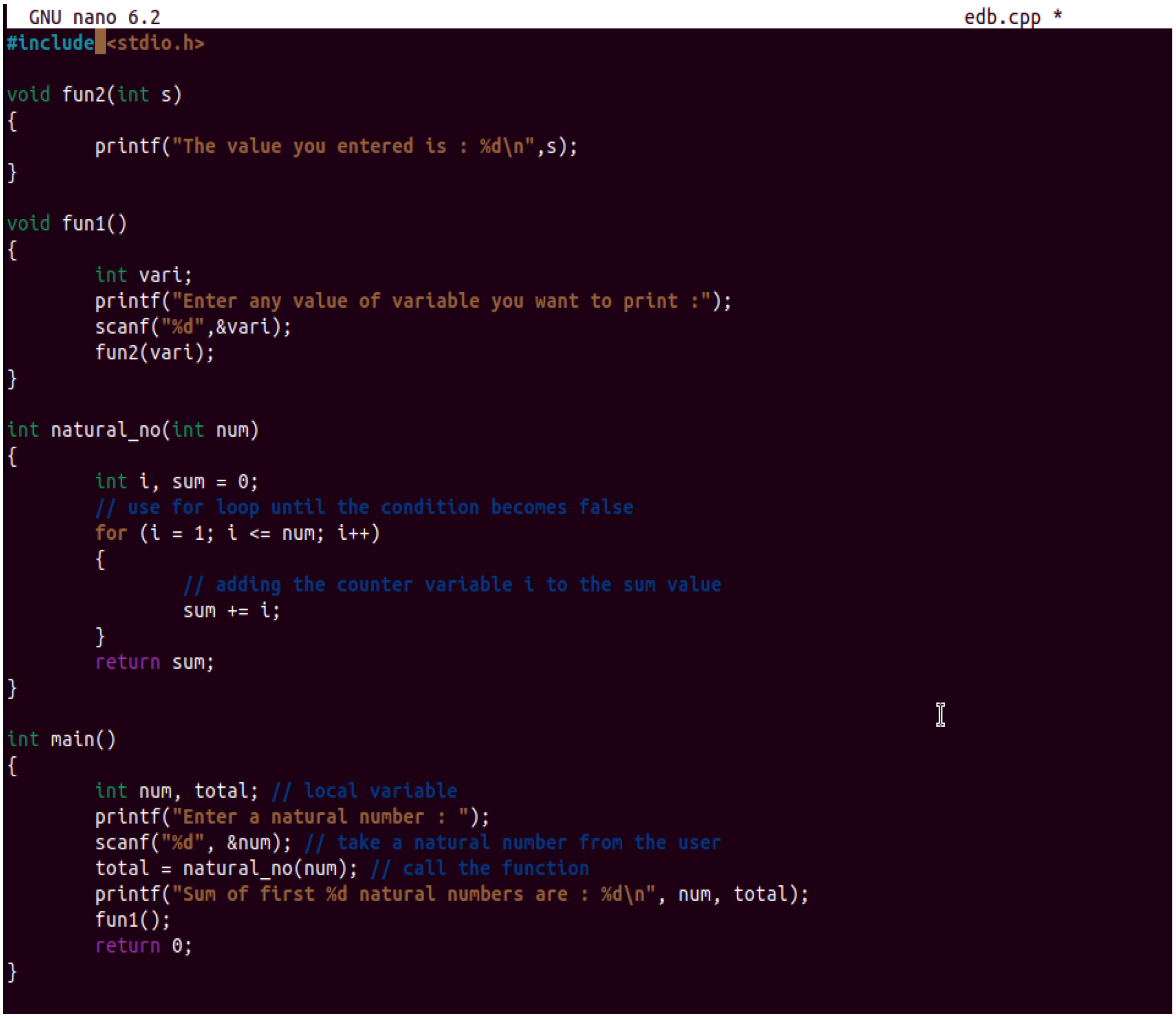


```
(gdb) break ignore.cpp:6
Breakpoint 1 at 0xaec: file ignore.cpp, line 6.
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000000aec in print(std::array<int, 10ul> const&) at ignore.cpp:6
(gdb) ignore 1 7
Will ignore next 7 crossings of breakpoint 1.
(gdb) r
`/home/ubuntu/ignore' has changed; re-reading symbols.
Starting program: /home/ubuntu/ignore
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".
nums [0] : 3
nums [1] : 4
nums [2] : 5
nums [3] : 6
nums [4] : 2
nums [5] : 7
nums [6] : 3

Breakpoint 1, print (nums=...) at ignore.cpp:6
6                       cout << "nums [" << i << "] : " << nums[i] << '\n';
(gdb) c
Continuing.
nums [7] : 8

Breakpoint 1, print (nums=...) at ignore.cpp:6
6                       cout << "nums [" << i << "] : " << nums[i] << '\n';
(gdb) c
Continuing.
nums [8] : 3

Breakpoint 1, print (nums=...) at ignore.cpp:6
6                       cout << "nums [" << i << "] : " << nums[i] << '\n';
(gdb) c
Continuing.
nums [9] : 10
[Inferior 1 (process 5073) exited normally]
(gdb)
```

We can see that gdb ignored the first 8 hits of the breakpoint by not stopping normal execution. After the first 8 hits were ignored, it resumed normal function of break-point from iteration 8.

11) Enable/disable a break-point:

Let's say we run the following program using gdb:

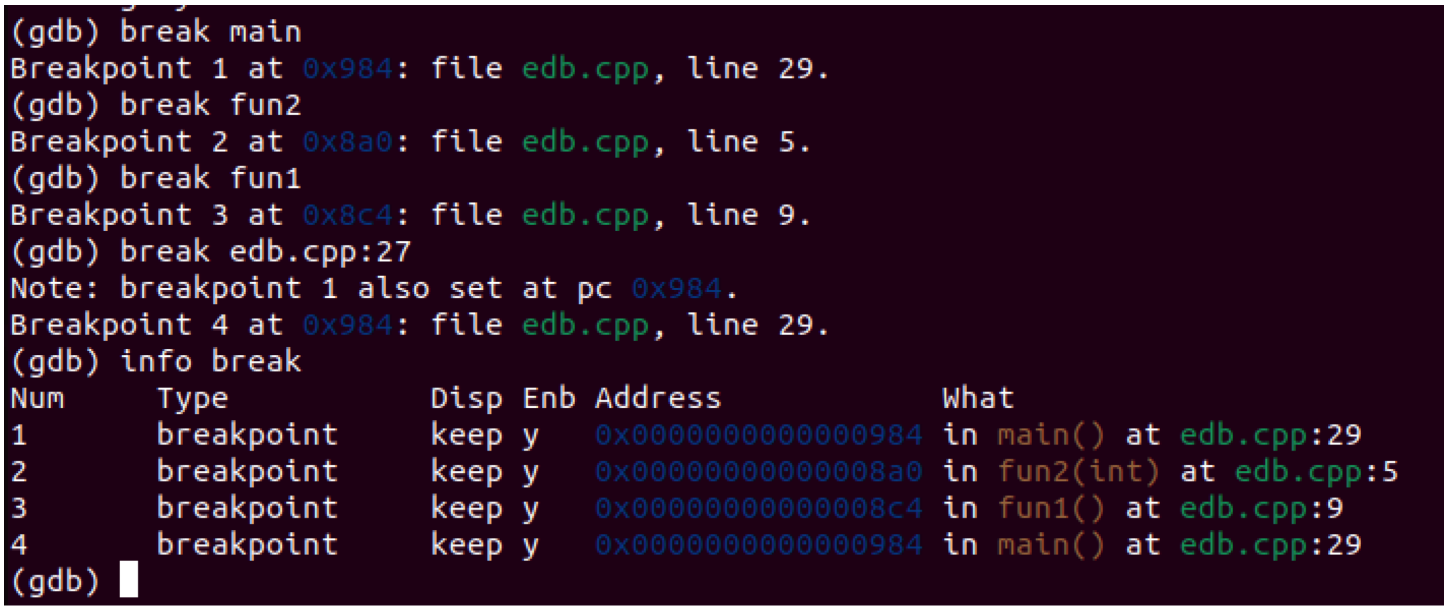


```
 GNU nano 6.2                                                          edb.cpp *
#include <stdio.h>

void fun2(int s)
{
        printf("The value you entered is : %d\n",s);
}

void fun1()
{
        int vari;
        printf("Enter any value of variable you want to print :");
        scanf("%d",&vari);
        fun2(vari);
}

int natural_no(int num)
{
        int i, sum = 0;
        // use for loop until the condition becomes false
        for (i = 1; i <= num; i++)
        {
                // adding the counter variable i to the sum value
                sum += i;
        }
        return sum;
}

int main()
{
        int num, total; // local variable
        printf("Enter a natural number : ");
        scanf("%d", &num); // take a natural number from the user
        total = natural_no(num); // call the function
        printf("Sum of first %d natural numbers are : %d\n", num, total);
        fun1();
        return 0;
}
```

This is basically a simple program where the user can get the sum of all natural numbers from 1 to the number entered and through the other functions the user gets to see the number entered.

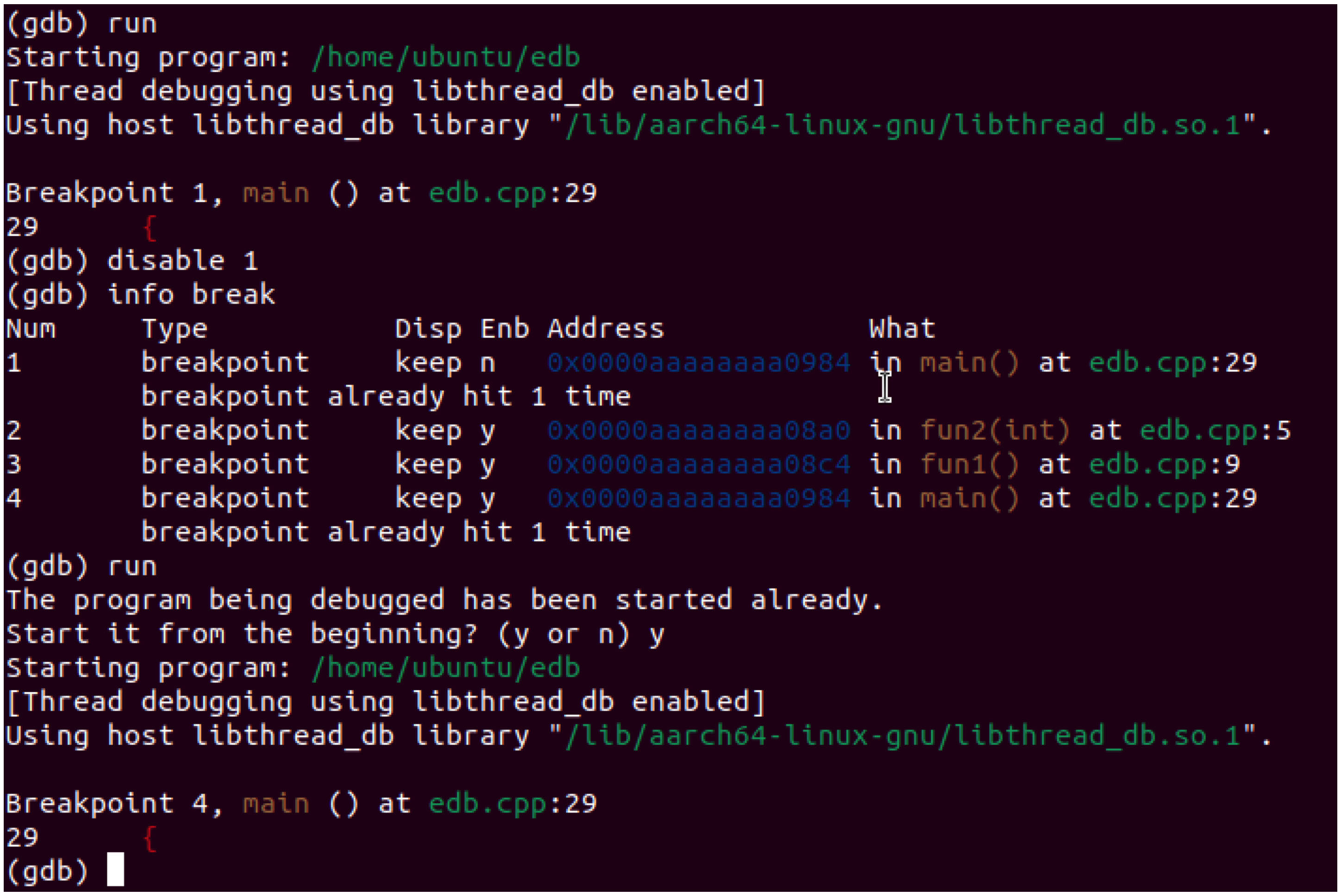
Now after going through Step -1 which says running a program through debugger:

a. Introducing Breakpoints and displaying break points with command ⇒ info break



```
(gdb) break main
Breakpoint 1 at 0x984: file edb.cpp, line 29.
(gdb) break fun2
Breakpoint 2 at 0x8a0: file edb.cpp, line 5.
(gdb) break fun1
Breakpoint 3 at 0x8c4: file edb.cpp, line 9.
(gdb) break edb.cpp:27
Note: breakpoint 1 also set at pc 0x984.
Breakpoint 4 at 0x984: file edb.cpp, line 29.
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000000984 in main() at edb.cpp:29
2       breakpoint     keep y   0x00000000000008a0 in fun2(int) at edb.cpp:5
3       breakpoint     keep y   0x00000000000008c4 in fun1() at edb.cpp:9
4       breakpoint     keep y   0x0000000000000984 in main() at edb.cpp:29
(gdb)
```

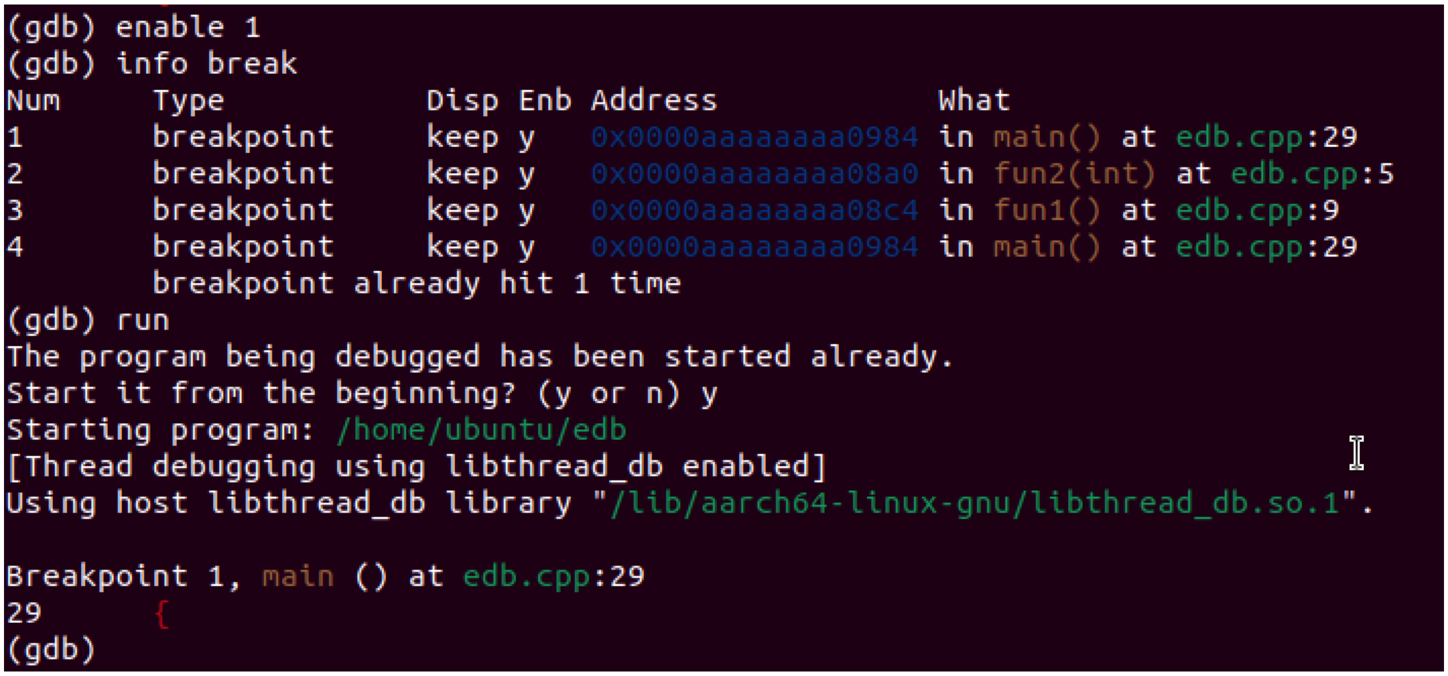
b. **Now, here we are Disabling the Breakpoint 1 ⇒ disable 1**



```
(gdb) run
Starting program: /home/ubuntu/edb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at edb.cpp:29
29        {
(gdb) disable 1
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep n   0x0000aaaaaaaa0984 in main() at edb.cpp:29
        breakpoint already hit 1 time
2       breakpoint     keep y   0x0000aaaaaaaa08a0 in fun2(int) at edb.cpp:5
3       breakpoint     keep y   0x0000aaaaaaaa08c4 in fun1() at edb.cpp:9
4       breakpoint     keep y   0x0000aaaaaaaa0984 in main() at edb.cpp:29
        breakpoint already hit 1 time
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ubuntu/edb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".

Breakpoint 4, main () at edb.cpp:29
29        {
(gdb)
```

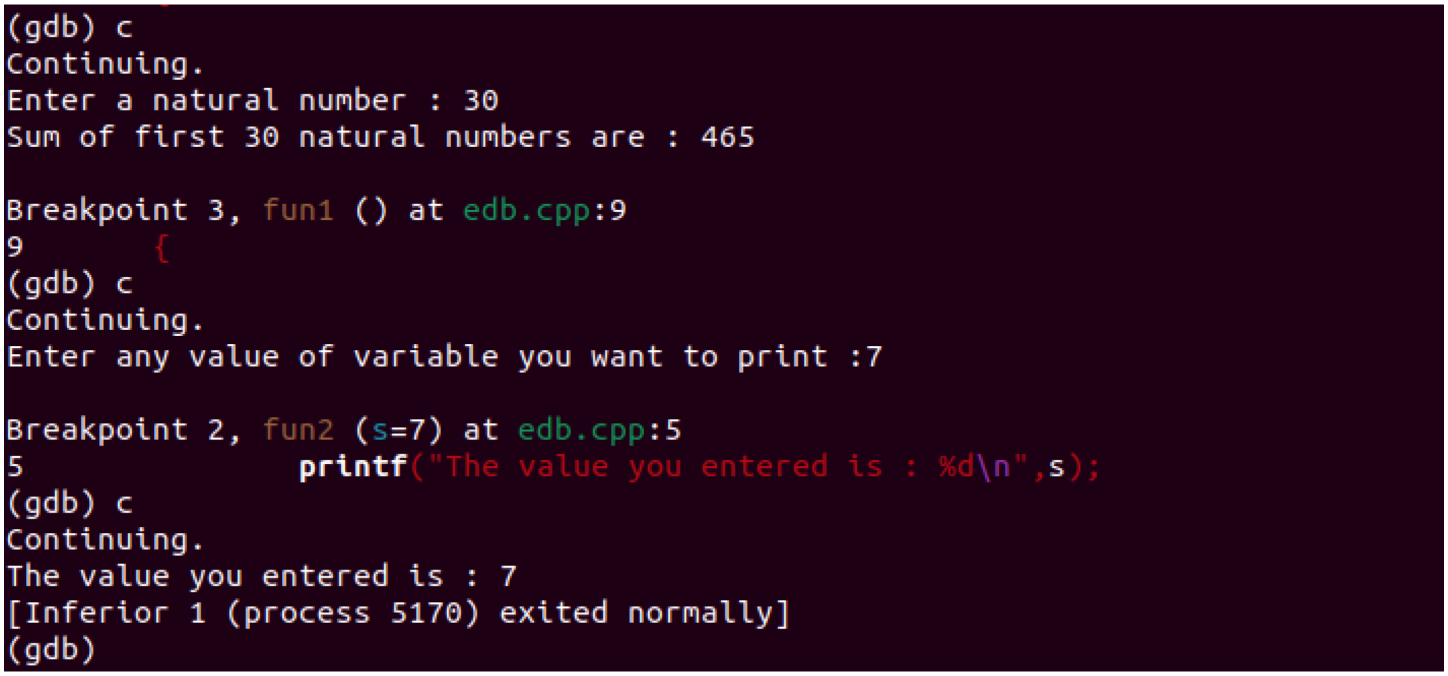
So, after disabling the breakpoint 1 when we do info break again we see 'n' under Enb which tells that breakpoint 1 is not enabled anymore. Also , again on running the program we see directly the program stops on Breakpoint 4 and not on Breakpoint 1.

c. **Here we are Enabling the Breakpoint 1 ⇒ enable 1**


```
(gdb) enable 1
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000aaaaaaaa0984 in main() at edb.cpp:29
2       breakpoint     keep y   0x0000aaaaaaaa08a0 in fun2(int) at edb.cpp:5
3       breakpoint     keep y   0x0000aaaaaaaa08c4 in fun1() at edb.cpp:9
4       breakpoint     keep y   0x0000aaaaaaaa0984 in main() at edb.cpp:29
        breakpoint already hit 1 time
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ubuntu/edb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at edb.cpp:29
29      {
(gdb)
```

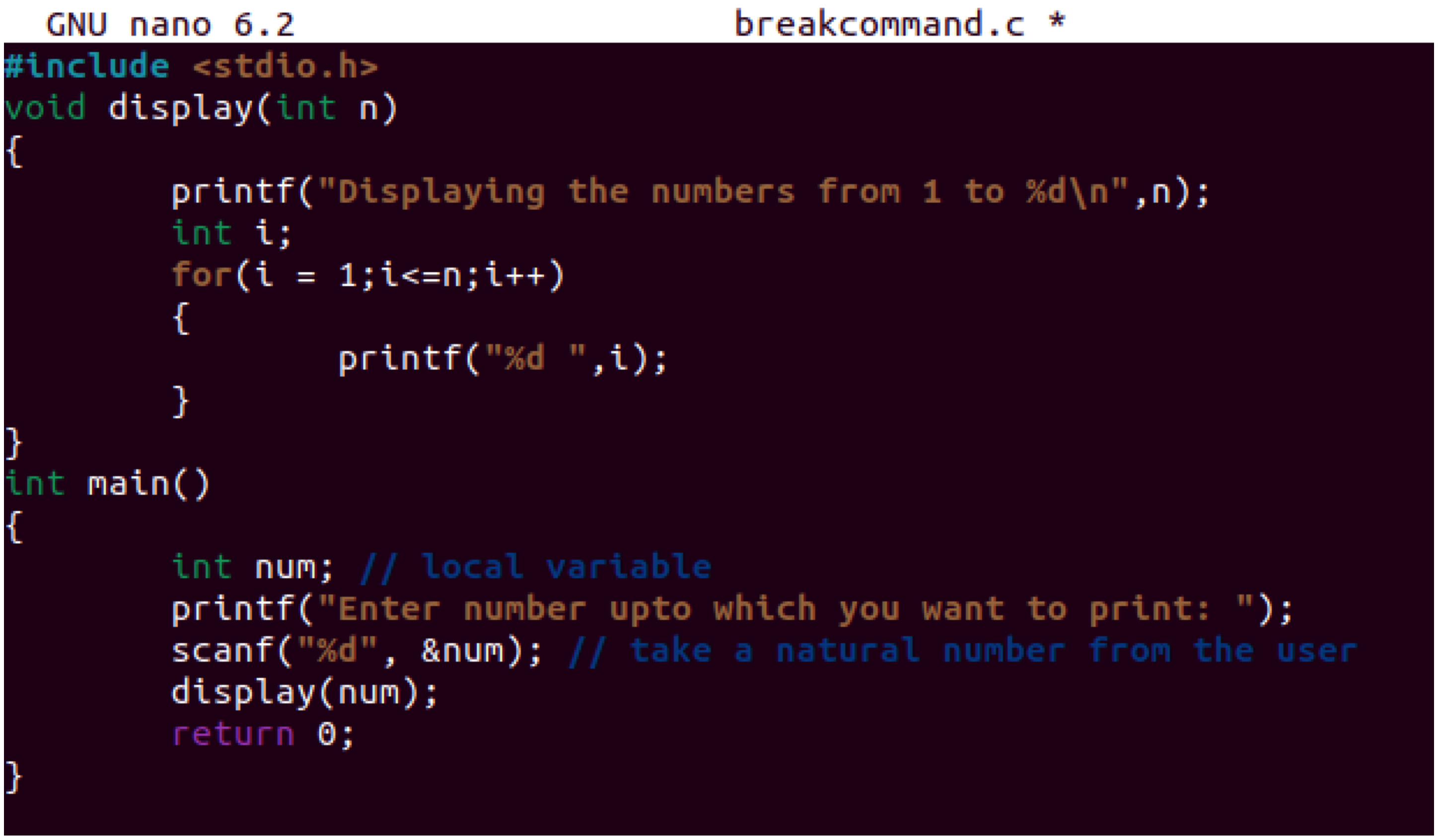
So, after enabling the breakpoint 1 when we do info break again we see ‘y’ under Enb which tells that breakpoint 1 is enabled. Also , again on running the program we see directly the program stops on Breakpoint 1.

d. **Lastly, we are running the whole program by pressing c (continue) when we encounter the Breakpoints.**


```
(gdb) c
Continuing.
Enter a natural number : 30
Sum of first 30 natural numbers are : 465

Breakpoint 3, fun1 () at edb.cpp:9
9       {
(gdb) c
Continuing.
Enter any value of variable you want to print :7

Breakpoint 2, fun2 (s=7) at edb.cpp:5
5               printf("The value you entered is : %d\n",s);
(gdb) c
Continuing.
The value you entered is : 7
[Inferior 1 (process 5170) exited normally]
(gdb)
```

12) Break condition and Command:

Let's say we run the following program using gdb:

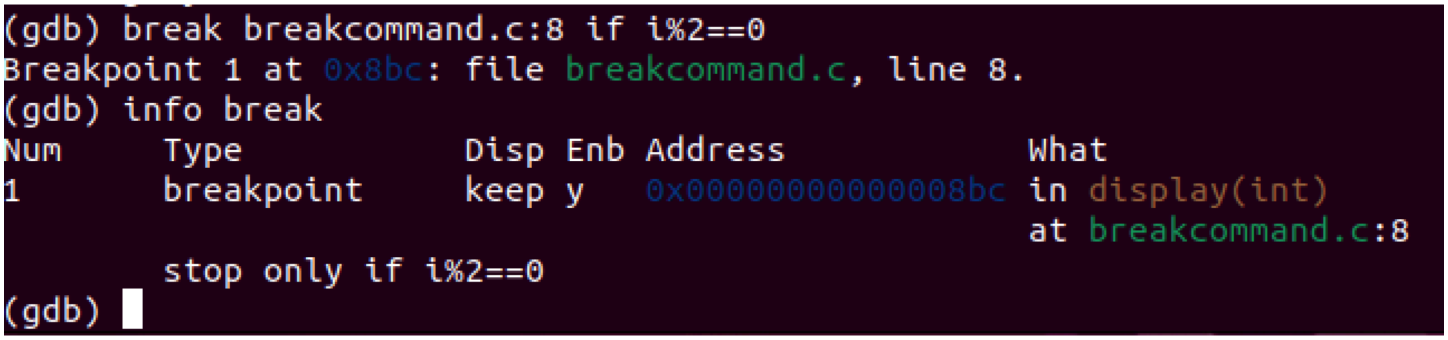


```
  GNU nano 6.2                          breakcommand.c *
#include <stdio.h>
void display(int n)
{
        printf("Displaying the numbers from 1 to %d\n",n);
        int i;
        for(i = 1;i<=n;i++)
        {
                printf("%d ",i);
        }
}
int main()
{
        int num; // local variable
        printf("Enter number upto which you want to print: ");
        scanf("%d", &num); // take a natural number from the user
        display(num);
        return 0;
}
```

This is basically a simple program where the user enters a number and gets all the numbers from 1 to the number entered.

Now after going through Step -1 which says running a program through debugger:

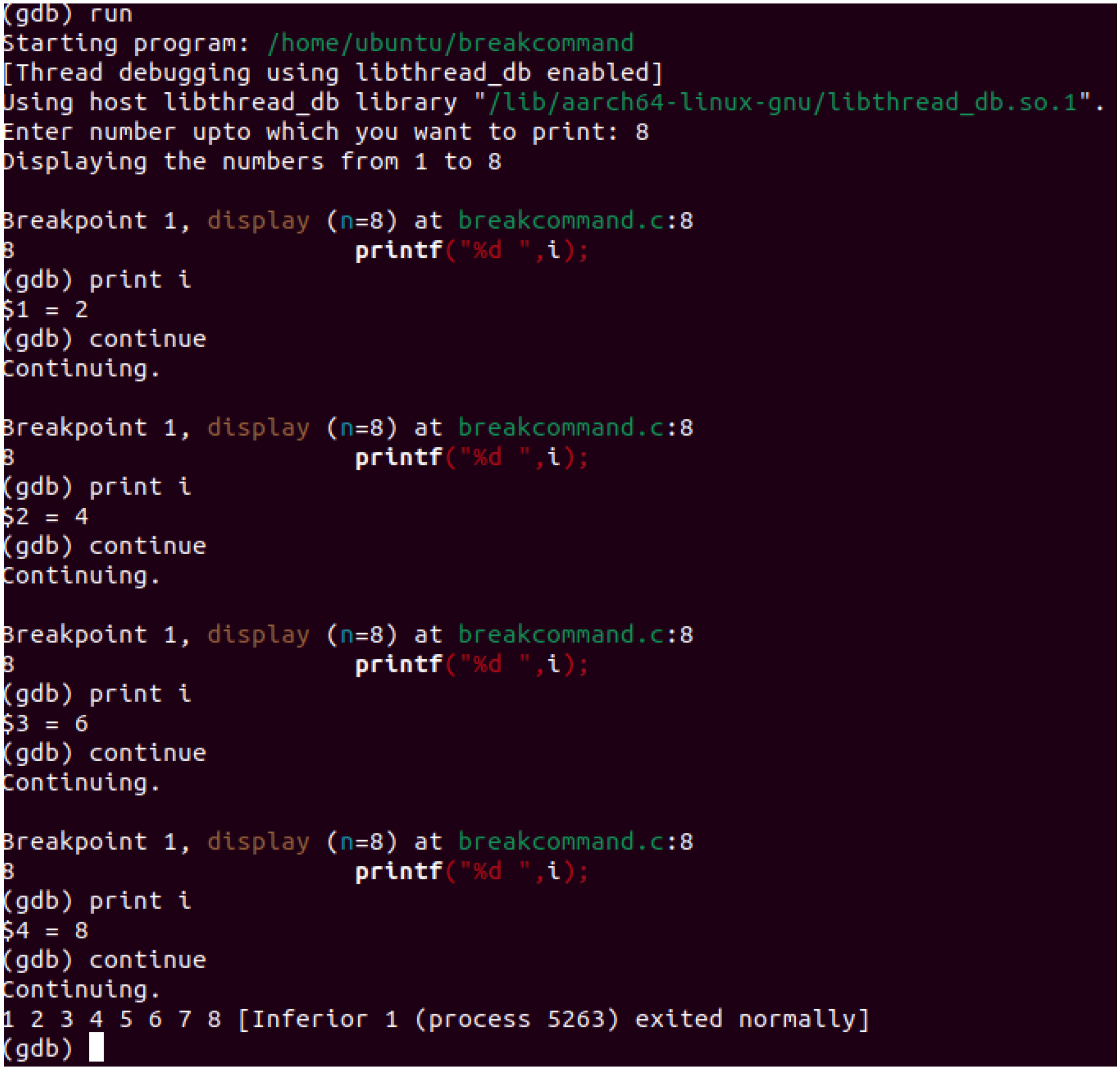
a. Introducing breakpoint in prog1.c at line 8 with condition if i%2 == 0 ⇒ break prog1.c:8 if i%2==0



```
(gdb) break breakcommand.c:8 if i%2==0
Breakpoint 1 at 0x8bc: file breakcommand.c, line 8.
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000000008bc in display(int)
                                                   at breakcommand.c:8
        stop only if i%2==0
(gdb) 
```

So, here what this command does is that whenever it gets a multiple of 2 (say even number) it causes a break or pause in the execution of the program.

b. Running the program and pressing c (continue) till we get the output.

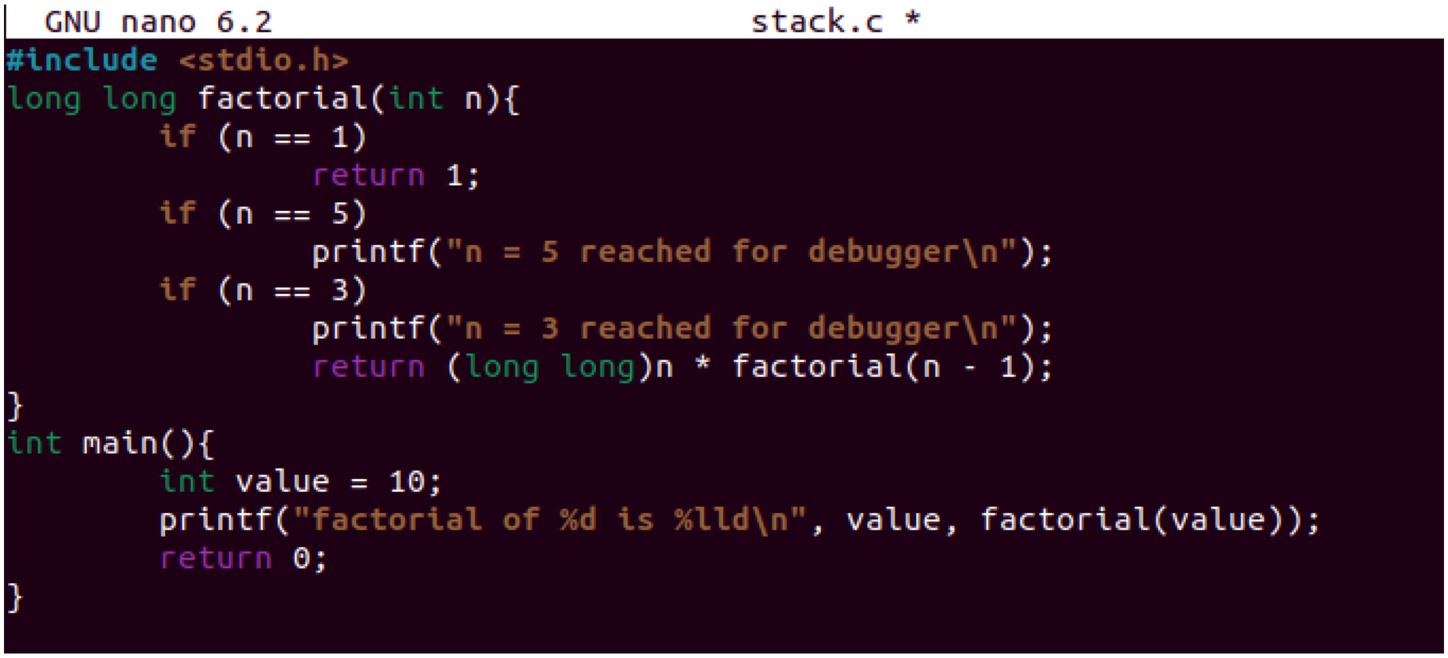


```
(gdb) run
Starting program: /home/ubuntu/breakcommand
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".
Enter number upto which you want to print: 8
Displaying the numbers from 1 to 8

Breakpoint 1, display (n=8) at breakcommand.c:8
8                       printf("%d ",i);
(gdb) print i
$1 = 2
(gdb) continue
Continuing.

Breakpoint 1, display (n=8) at breakcommand.c:8
8                       printf("%d ",i);
(gdb) print i
$2 = 4
(gdb) continue
Continuing.

Breakpoint 1, display (n=8) at breakcommand.c:8
8                       printf("%d ",i);
(gdb) print i
$3 = 6
(gdb) continue
Continuing.

Breakpoint 1, display (n=8) at breakcommand.c:8
8                       printf("%d ",i);
(gdb) print i
$4 = 8
(gdb) continue
Continuing.
1 2 3 4 5 6 7 8 [Inferior 1 (process 5263) exited normally]
(gdb) 
```

So, on pressing run we get the breakpoints whenever i gets a value which is a multiple of 2.
Thus if n = 8, we get four breakpoints at 2 , 4, 6 and 8.
On pressing print i and then c (continue) we can get to see the respective values of i too which are respectively 2,4,6 and 8 as specified already.

13) Examining stack trace:

Let’s define a recursive function to see how the stack trace examination works in the gdb

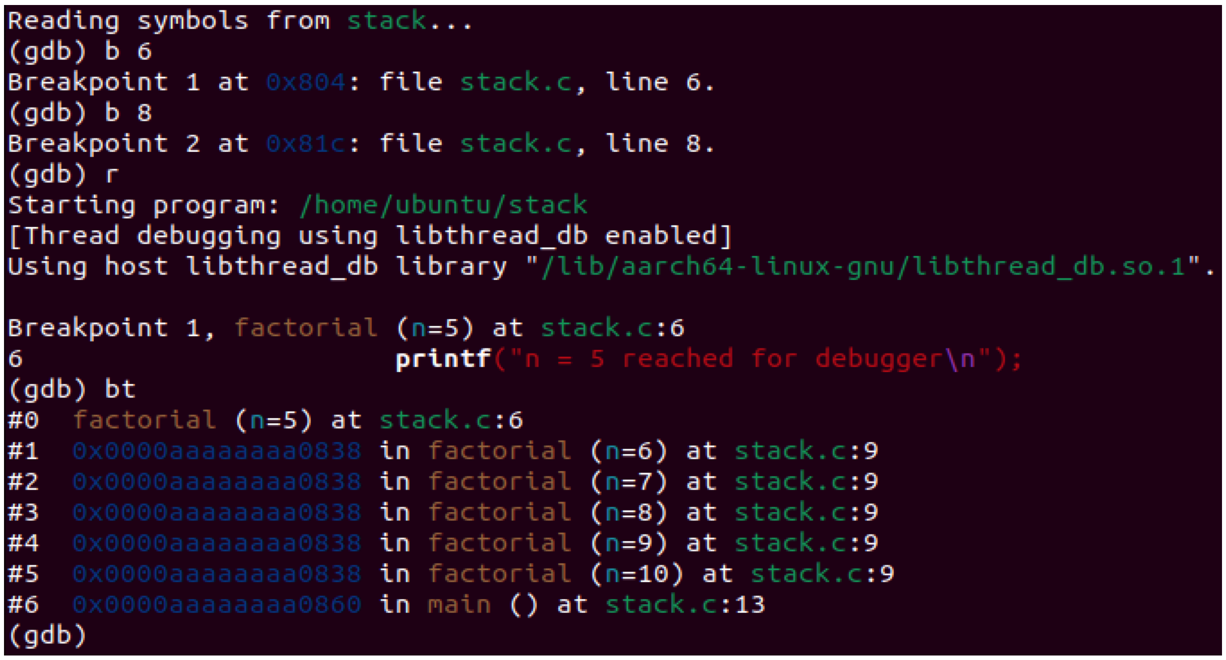


```
  GNU nano 6.2                                    stack.c *
#include <stdio.h>
long long factorial(int n){
        if (n == 1)
                return 1;
        if (n == 5)
                printf("n = 5 reached for debugger\n");
        if (n == 3)
                printf("n = 3 reached for debugger\n");
                return (long long)n * factorial(n - 1);
}
int main(){
        int value = 10;
        printf("factorial of %d is %lld\n", value, factorial(value));
        return 0;
}
```

This is a standard factorial recursive function which can overshoot for large value of n

Let's compile and run it it with gdb

1. compile the program “gcc stack.c -o stack -g”
2. run the program in gdb: “gdb stack”
3. add breakpoints to the line 6 and 8 (the printf in the factorial function): “b 6” and “b 8”
4. check if breakpoints are set or not: “info b”
5. run the program till a breakpoint occurs: “r”
    a. Now the program will stop at the first breakpoint (at line 6)
6. examine the stack trace: “bt”
    a. Output from my side:



```
Reading symbols from stack...
(gdb) b 6
Breakpoint 1 at 0x804: file stack.c, line 6.
(gdb) b 8
Breakpoint 2 at 0x81c: file stack.c, line 8.
(gdb) r
Starting program: /home/ubuntu/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".

Breakpoint 1, factorial (n=5) at stack.c:6
6                       printf("n = 5 reached for debugger\n");
(gdb) bt
#0  factorial (n=5) at stack.c:6
#1  0x0000aaaaaaaa0838 in factorial (n=6) at stack.c:9
#2  0x0000aaaaaaaa0838 in factorial (n=7) at stack.c:9
#3  0x0000aaaaaaaa0838 in factorial (n=8) at stack.c:9
#4  0x0000aaaaaaaa0838 in factorial (n=9) at stack.c:9
#5  0x0000aaaaaaaa0838 in factorial (n=10) at stack.c:9
#6  0x0000aaaaaaaa0860 in main () at stack.c:13
(gdb)
```

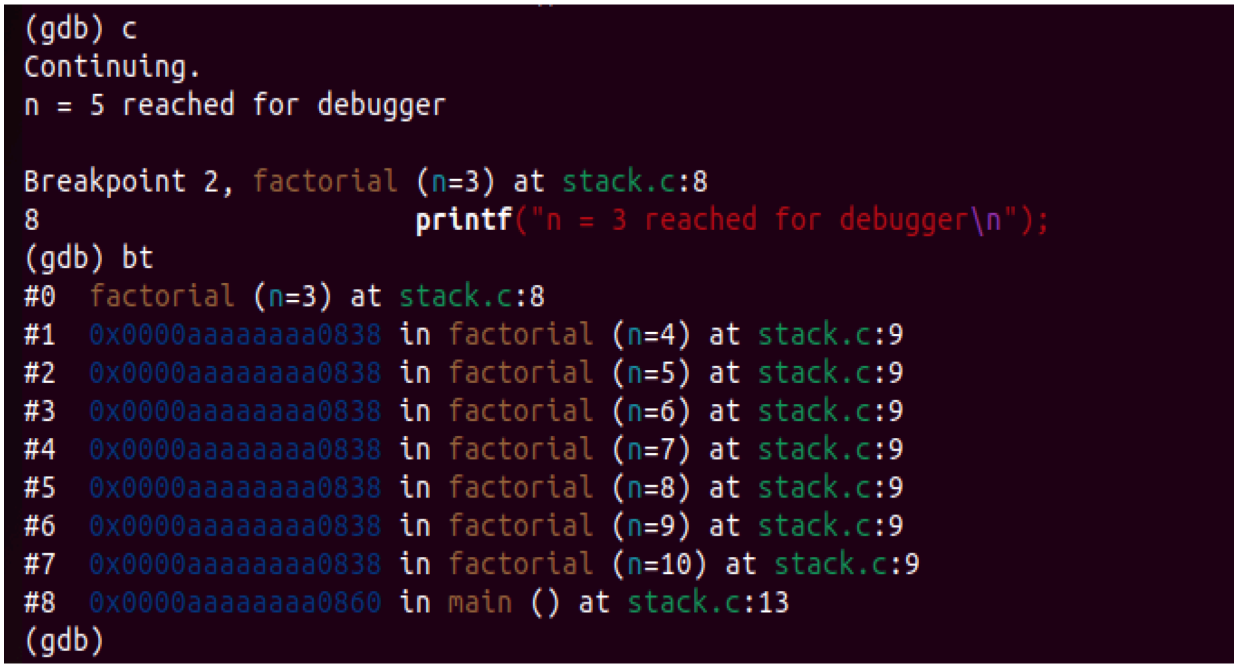
b. Here we can see the stack trace of the program in execution, the factorial function is called from main, then it is recursively called till the breakpoint at n = 5

7. continue till next breakpoint: “c”

a. Now the program will stop again at breakpoint of line 8

8. examine the stack trace again: “bt”

a. output from my side:



```
(gdb) c
Continuing.
n = 5 reached for debugger

Breakpoint 2, factorial (n=3) at stack.c:8
8                       printf("n = 3 reached for debugger\n");
(gdb) bt
#0  factorial (n=3) at stack.c:8
#1  0x0000aaaaaaaa0838 in factorial (n=4) at stack.c:9
#2  0x0000aaaaaaaa0838 in factorial (n=5) at stack.c:9
#3  0x0000aaaaaaaa0838 in factorial (n=6) at stack.c:9
#4  0x0000aaaaaaaa0838 in factorial (n=7) at stack.c:9
#5  0x0000aaaaaaaa0838 in factorial (n=8) at stack.c:9
#6  0x0000aaaaaaaa0838 in factorial (n=9) at stack.c:9
#7  0x0000aaaaaaaa0838 in factorial (n=10) at stack.c:9
#8  0x0000aaaaaaaa0860 in main () at stack.c:13
(gdb)
```

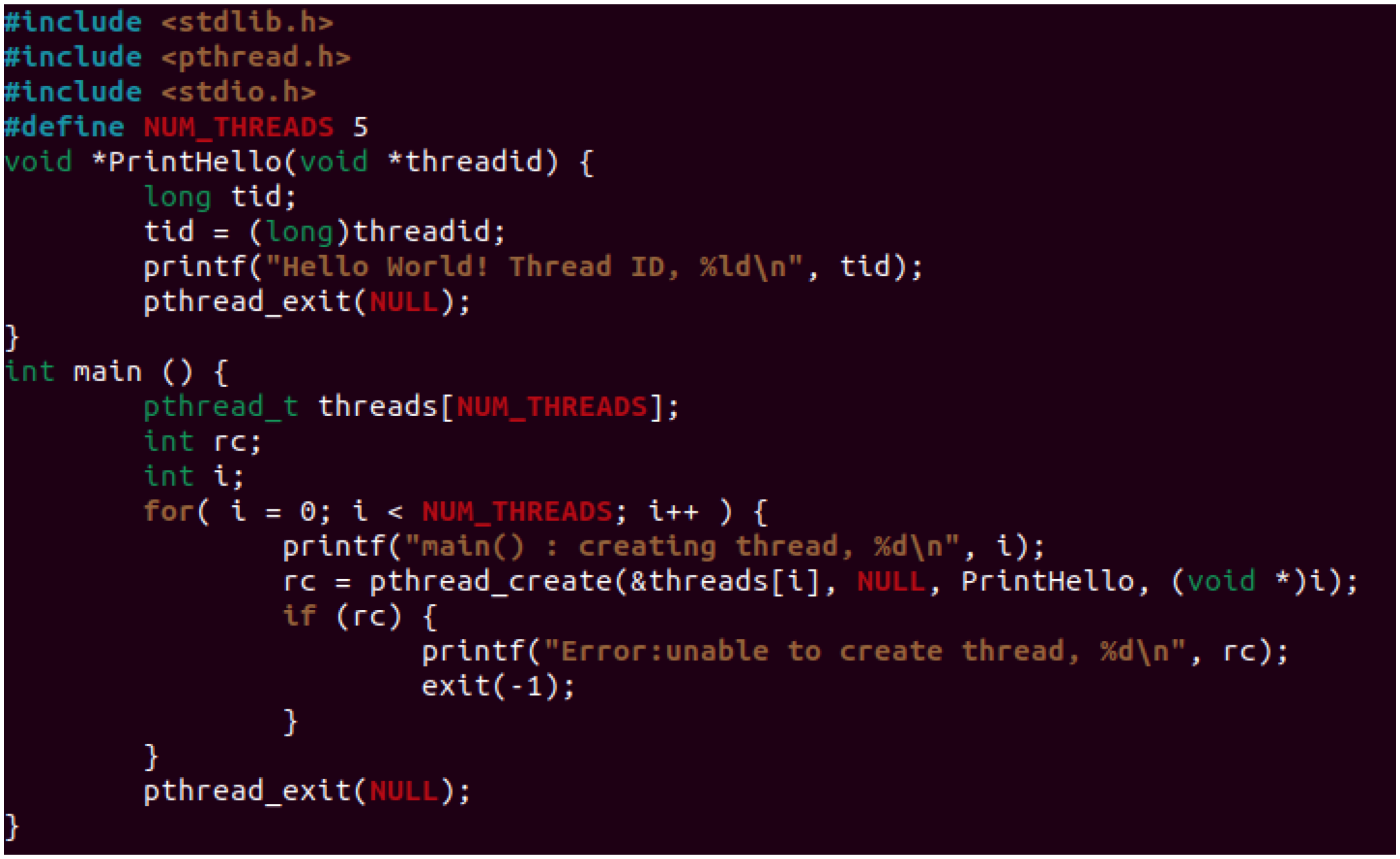
b. Here we see that the previous stack trace grew from n = 5 to n = 3, as expected

9. continue till the next breakpoint (there aren't any so the program will complete executing) : "c"

10. exit from gdb: "q"

14) Examining stack trace for multi-threaded program:

Consider the following simple program for this part



```
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *PrintHello(void *threadid) {
        long tid;
        tid = (long)threadid;
        printf("Hello World! Thread ID, %ld\n", tid);
        pthread_exit(NULL);
}
int main () {
        pthread_t threads[NUM_THREADS];
        int rc;
        int i;
        for( i = 0; i < NUM_THREADS; i++ ) {
                printf("main() : creating thread, %d\n", i);
                rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);
                if (rc) {
                        printf("Error:unable to create thread, %d\n", rc);
                        exit(-1);
                }
        }
        pthread_exit(NULL);
}
```

Here the main calls five threads and they print hello and exit

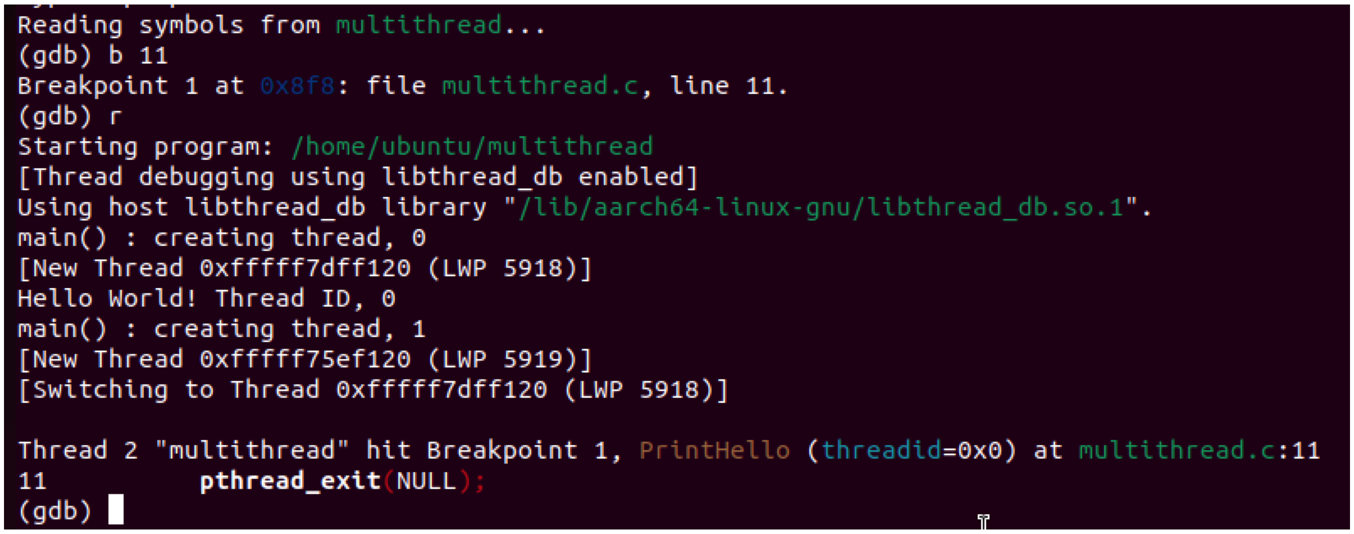
1. compile the program: "gcc multithread.c -o multithread -g"

2. open program in gdb: "gdb multithread"

3. add a breakpoint where we can stop the processes: "b 10"

4. start the program: “r”

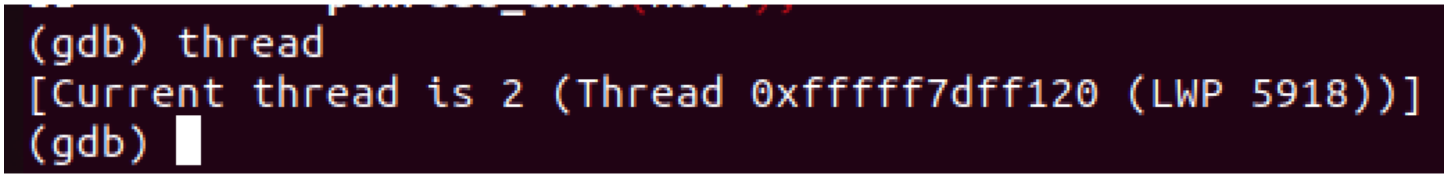
a. we see the gdb notes that main() is creating threads



```
Reading symbols from multithread...
(gdb) b 11
Breakpoint 1 at 0x8f8: file multithread.c, line 11.
(gdb) r
Starting program: /home/ubuntu/multithread
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".
main() : creating thread, 0
[New Thread 0xffffff7dff120 (LWP 5918)]
Hello World! Thread ID, 0
main() : creating thread, 1
[New Thread 0xffffff75ef120 (LWP 5919)]
[Switching to Thread 0xffffff7dff120 (LWP 5918)]

Thread 2 "multithread" hit Breakpoint 1, PrintHello (threadid=0x0) at multithread.c:11
11          pthread_exit(NULL);
(gdb) 
```

b. Now the program will stop when the threads are running

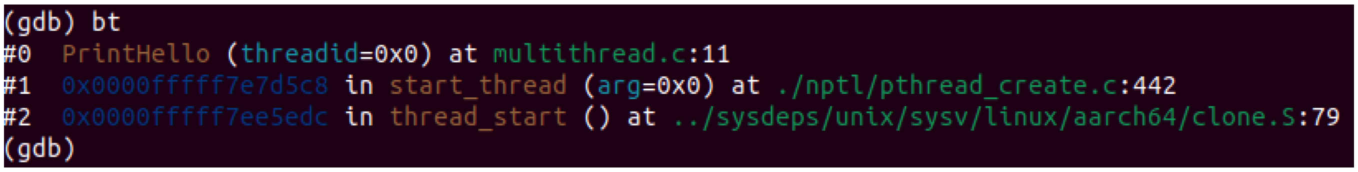
5. see the current thread: “thread”



```
(gdb) thread
[Current thread is 2 (Thread 0xffffff7dff120 (LWP 5918))]
(gdb) 
```

6. see the backtrace of the current thread: “bt”

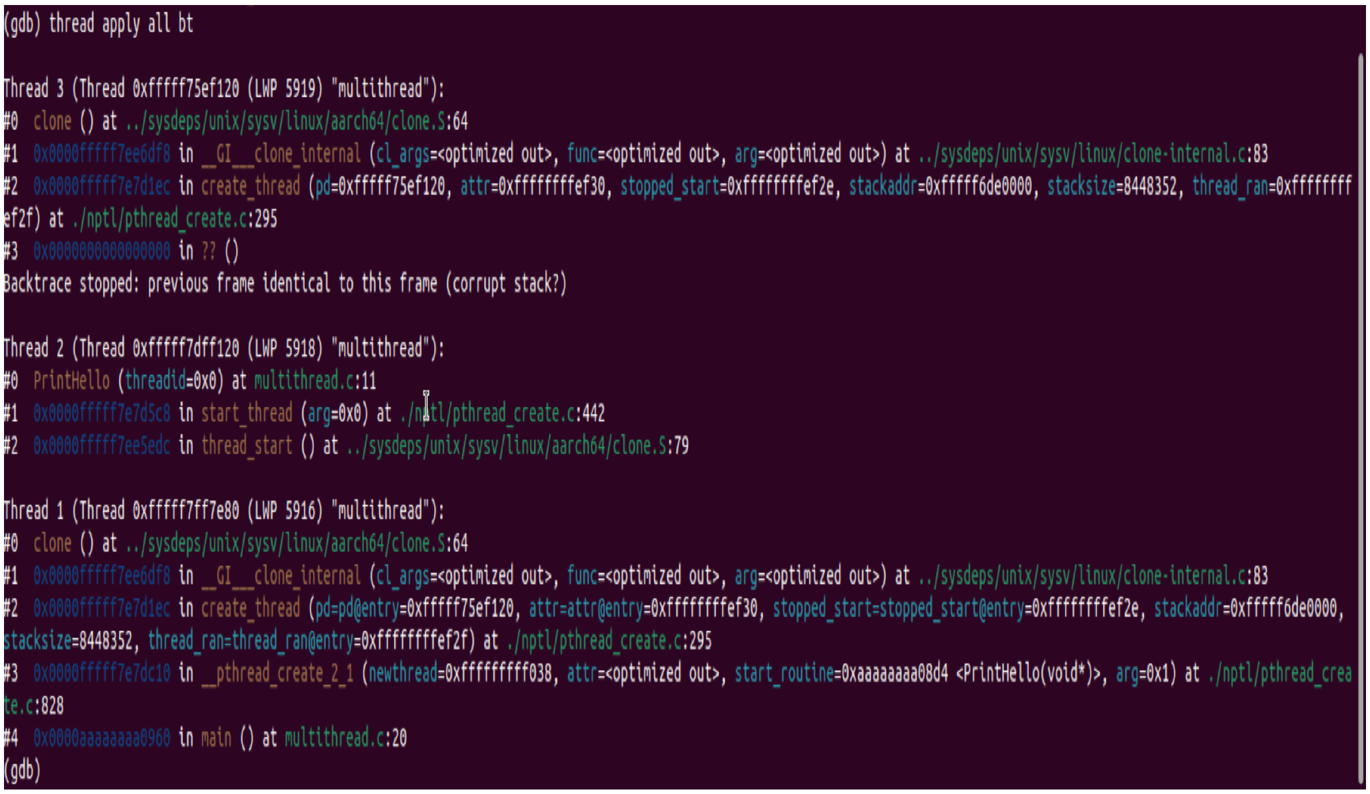
a. we see that the thread is not back tracing to main, it is back tracing to start_thread implemented in the OS itself



```
(gdb) bt
#0  PrintHello (threadid=0x0) at multithread.c:11
#1  0x0000fffff7e7d5c8 in start_thread (arg=0x0) at ./nptl/pthread_create.c:442
#2  0x0000fffff7ee5edc in thread_start () at ../sysdeps/unix/sysv/linux/aarch64/clone.S:79
(gdb)
```

7. see the backtrace of all the processes: “thread apply all bt”

a. here we can see that thread 1 is from main making all the threads (was only able to make two threads before thread 2 stopped), and thread 3 has been made but hasn’t started to execute PrintHello() just yet



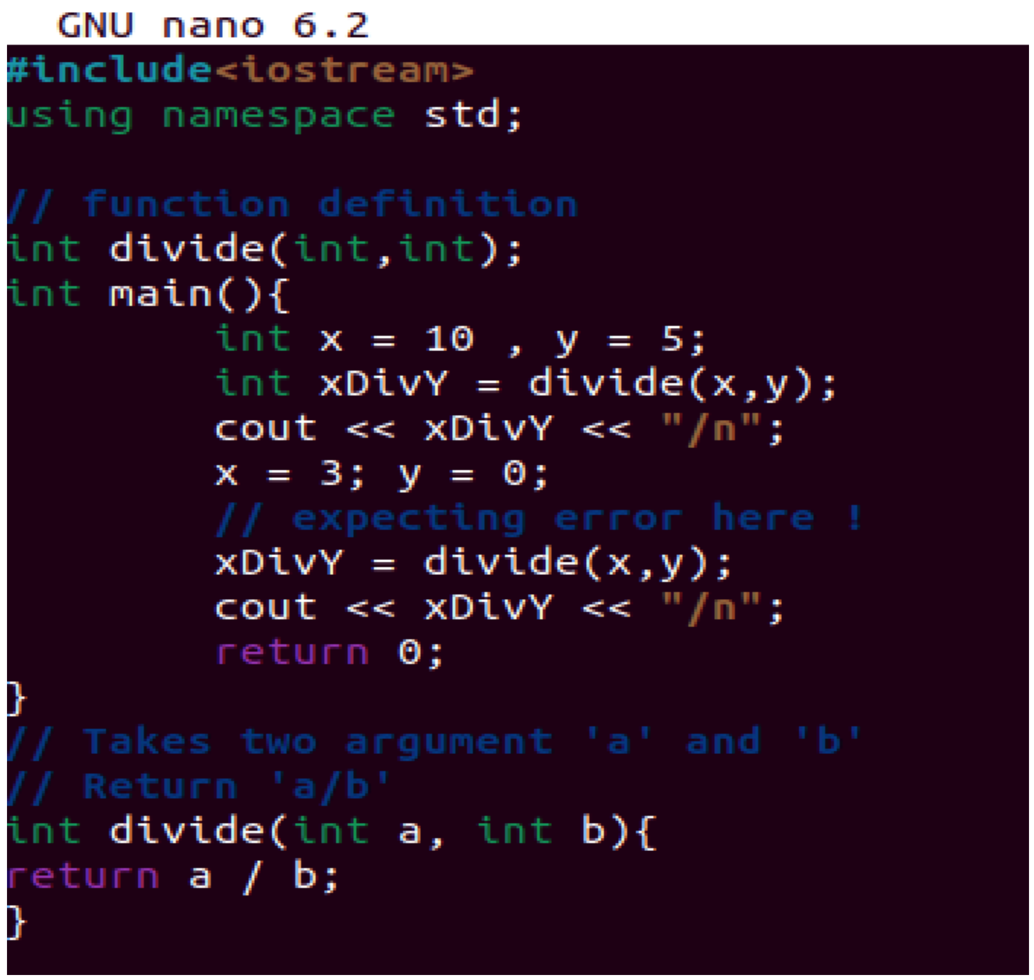
```
(gdb) thread apply all bt

Thread 3 (Thread 0xffffff75ef120 (LWP 5919) "multithread"):
#0  clone () at ../sysdeps/unix/sysv/linux/aarch64/clone.S:64
#1  0x0000fffff7ee6df8 in __GI___clone_internal (cl_args=<optimized out>, func=<optimized out>, arg=<optimized out>) at ../sysdeps/unix/sysv/linux/clone-internal.c:83
#2  0x0000fffff7e7d1ec in create_thread (pd=0xfffff75ef120, attr=0xffffffffef30, stopped_start=0xffffffffef2e, stackaddr=0xfffff6de0000, stacksize=8448352, thread_ran=0xffffffff
ef2f) at ./nptl/pthread_create.c:295
#3  0x0000000000000000 in ?? ()
Backtrace stopped: previous frame identical to this frame (corrupt stack?)

Thread 2 (Thread 0xfffff7dff120 (LWP 5918) "multithread"):
#0  PrintHello (threadid=0x0) at multithread.c:11
#1  0x0000fffff7e7d5c8 in start_thread (arg=0x0) at ./nptl/pthread_create.c:442
#2  0x0000fffff7ee5edc in thread_start () at ../sysdeps/unix/sysv/linux/aarch64/clone.S:79

Thread 1 (Thread 0xfffff7ff7e80 (LWP 5916) "multithread"):
#0  clone () at ../sysdeps/unix/sysv/linux/aarch64/clone.S:64
#1  0x0000fffff7ee6df8 in __GI___clone_internal (cl_args=<optimized out>, func=<optimized out>, arg=<optimized out>) at ../sysdeps/unix/sysv/linux/clone-internal.c:83
#2  0x0000fffff7e7d1ec in create_thread (pd=pd@entry=0xfffff75ef120, attr=attr@entry=0xffffffffef30, stopped_start=stopped_start@entry=0xffffffffef2e, stackaddr=0xfffff6de0000,
stacksize=8448352, thread_ran=thread_ran@entry=0xffffffffef2f) at ./nptl/pthread_create.c:295
#3  0x0000fffff7e7dc10 in __pthread_create_2_1 (newthread=0xfffffffff038, attr=<optimized out>, start_routine=0xaaaaaaaa08d4 <PrintHello(void*)>, arg=0x1) at ./nptl/pthread_crea
te.c:828
#4  0x0000aaaaaaaa0960 in main () at multithread.c:20
(gdb)
```

8. now you can exit or keep running the program till execution ends.

15) Core file debugging:

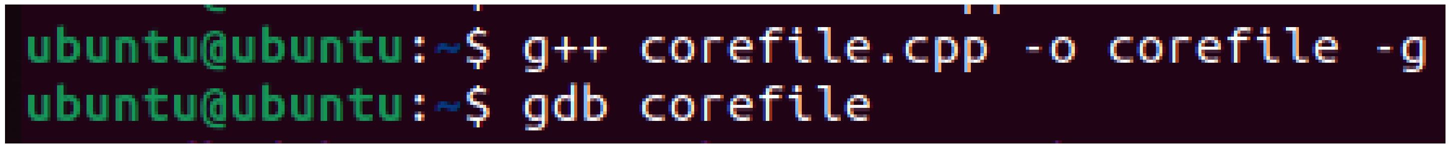
A "core dump" is a snapshot of memory at the instant the program crashes, typically saved in a file called "core". GDB can read the core dump and give you the line number of the crash, the arguments that were passed, and more.

Let's write a simple program to generate a core dump and debug it.



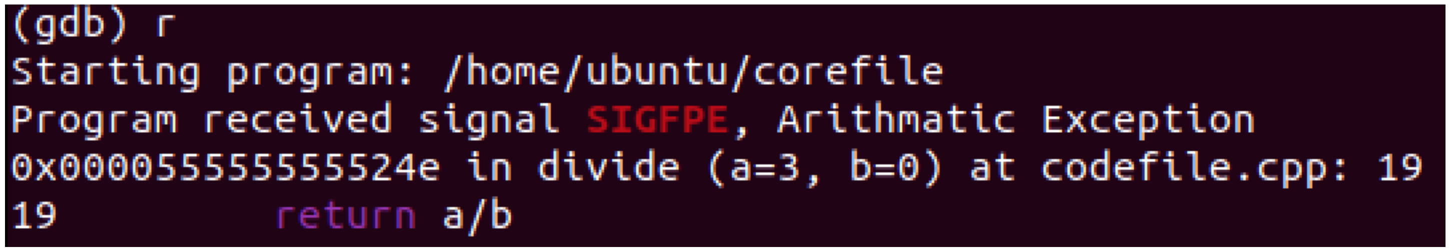
```
  GNU nano 6.2
#include<iostream>
using namespace std;

// function definition
int divide(int,int);
int main(){
        int x = 10 , y = 5;
        int xDivY = divide(x,y);
        cout << xDivY << "/n";
        x = 3; y = 0;
        // expecting error here !
        xDivY = divide(x,y);
        cout << xDivY << "/n";
        return 0;
}
// Takes two argument 'a' and 'b'
// Return 'a/b'
int divide(int a, int b){
return a / b;
}
```

1. We will first compile this program using g++ . Keep in mind to use -g option to debug with help of GDB. After that , we will load our executable file using gdb.



```
ubuntu@ubuntu:~$ g++ corefile.cpp -o corefile -g
ubuntu@ubuntu:~$ gdb corefile
```

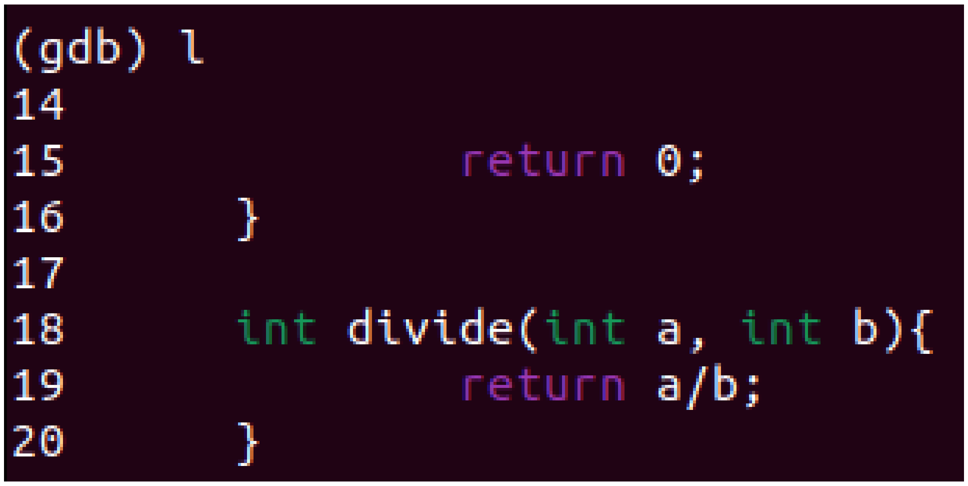
2. With r as the command, run the executable. Since there are not any breakpoints, it will run complete code.



```
(gdb) r
Starting program: /home/ubuntu/corefile
Program received signal SIGFPE, Arithmatic Exception
0x000055555555524e in divide (a=3, b=0) at codefile.cpp: 19
19          return a/b
```

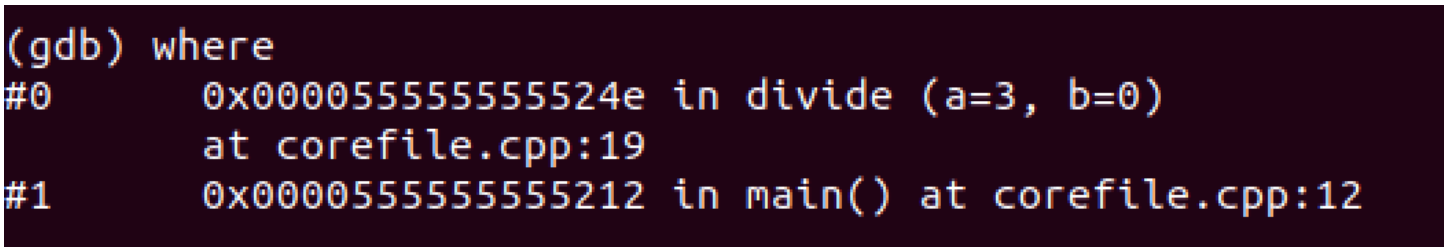
gdb is showing some information about core-file. It is showing that our program has received the signal Floating Point Error. It is also showing the line at which an exception takes place. It is on line 19 , in function divide where arguments are a = 3 and b = 0 respectively. It is also listing line number 19, “return a/b”.

3. We can print out context (code) where exceptions (/core file) occur using ‘l’ command.



```
(gdb) l
14
15                  return 0;
16          }
17
18          int divide(int a, int b){
19                  return a/b;
20          }
```

4. We can print the Stack trace using the “where” command.

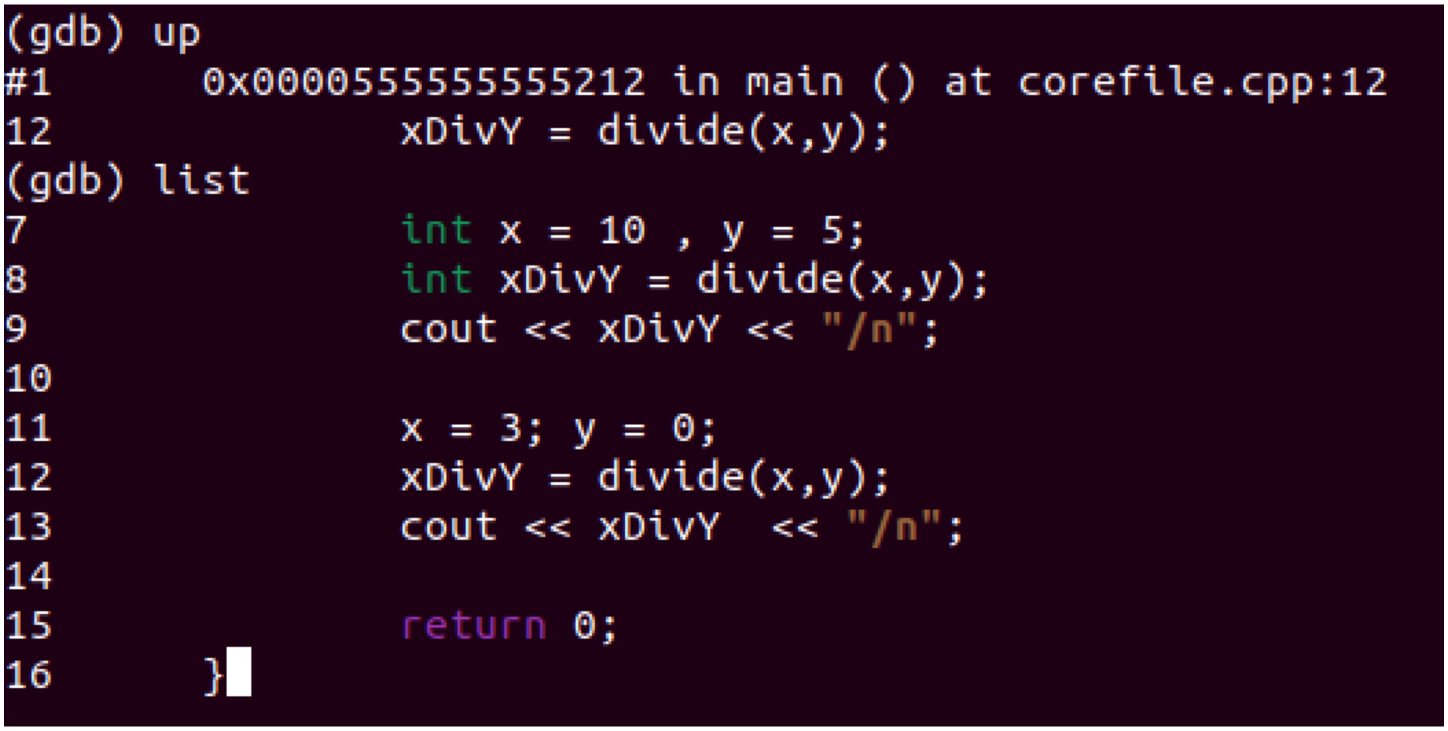


```
(gdb) where
#0      0x000055555555524e in divide (a=3, b=0)
        at corefile.cpp:19
#1      0x0000555555555212 in main() at corefile.cpp:12
```

This can be read as: The crash occurred in the function divide at line 19 of corefile.cpp. This, in turn, was called from the function main at line 13 of corefile.cpp.

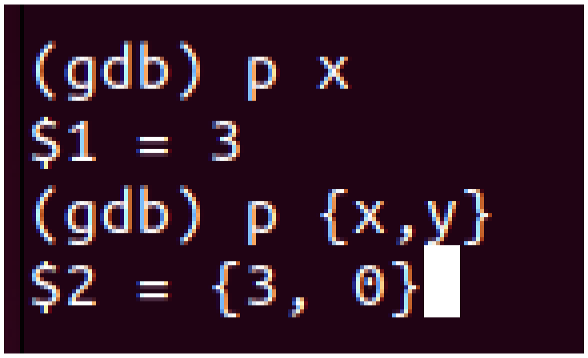
5. We can use the up and the down command to move from the current level of stack trace up or down one level. Here, we are going from default level ‘0’ to one level up using up.

We are also using list to list the code near to command from where the divide function is called.



```
(gdb) up
#1      0x0000555555555212 in main () at corefile.cpp:12
12              xDivY = divide(x,y);
(gdb) list
7               int x = 10 , y = 5;
8               int xDivY = divide(x,y);
9               cout << xDivY << "/n";
10
11              x = 3; y = 0;
12              xDivY = divide(x,y);
13              cout << xDivY  << "/n";
14
15              return 0;
16      }
```

6. We can print the current level of the stack using print command.



```
(gdb) p x
$1 = 3
(gdb) p {x,y}
$2 = {3, 0}
```

16) Debugging of an already running program:

When a program has not started as a process yet, we can attach it with gdb and then run it using the r command.

gdb program

But if the program is already running as a process then we have to perform following steps in order to attach gdb with it :

1. **Find the process id (pid) with the help of pidof command:**

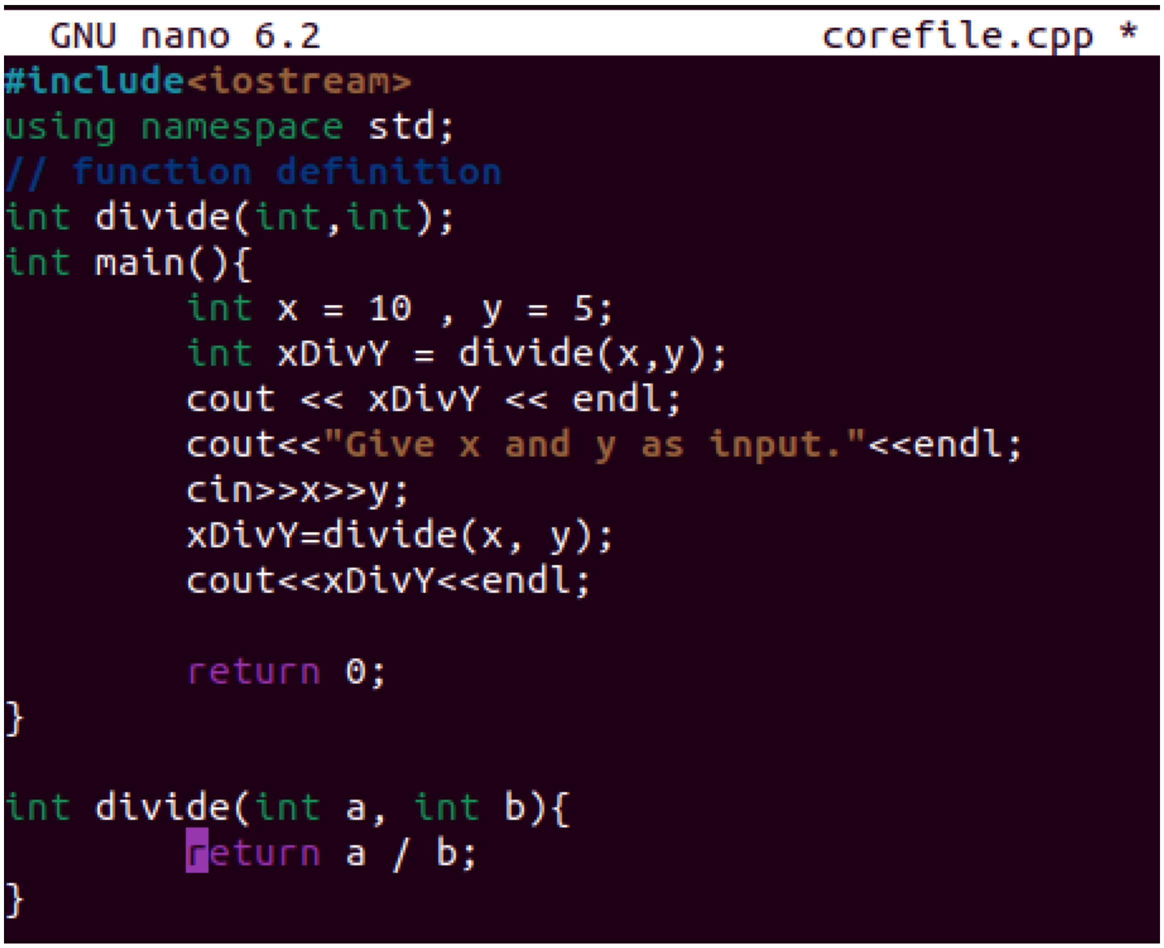
pidof program
# Replace program with a file name or path to the program.

2. **Attach GDB to this process:**

gdb program -p pid
# Replace program with a file name or path to the program, replace pid with an actual process id number from the ps output.

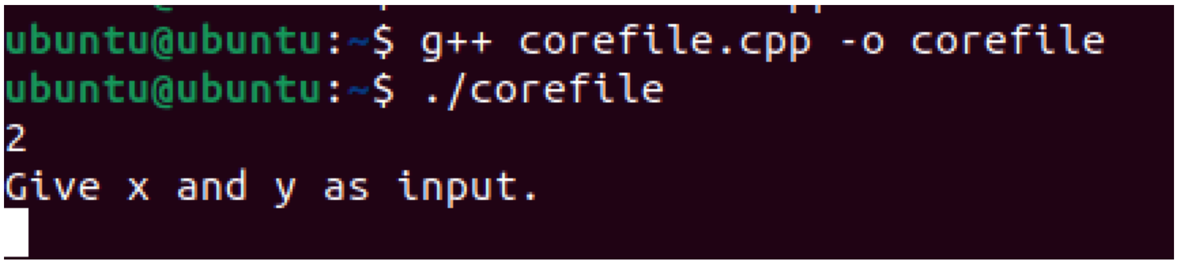
Example:

a. **Code**



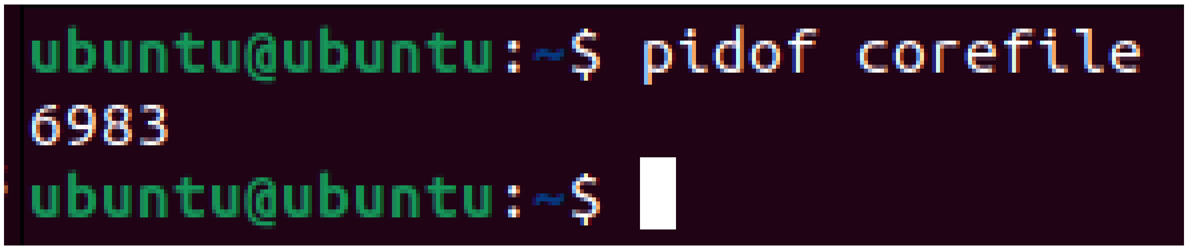
```
  GNU nano 6.2                              corefile.cpp *
#include<iostream>
using namespace std;
// function definition
int divide(int,int);
int main(){
        int x = 10 , y = 5;
        int xDivY = divide(x,y);
        cout << xDivY << endl;
        cout<<"Give x and y as input."<<endl;
        cin>>x>>y;
        xDivY=divide(x, y);
        cout<<xDivY<<endl;

        return 0;
}

int divide(int a, int b){
        return a / b;
}
```

b. **Process start running**



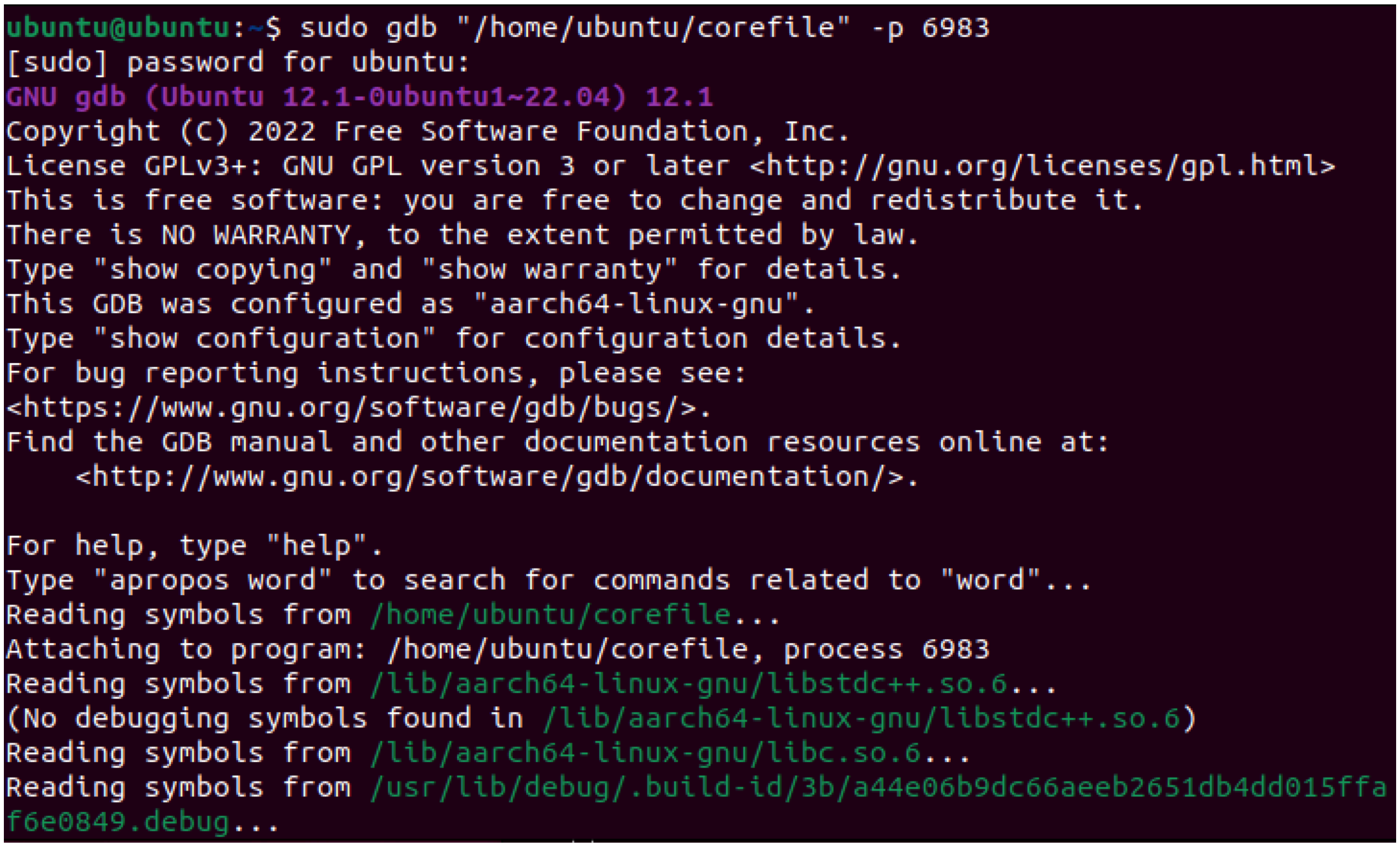
```
ubuntu@ubuntu:~$ g++ corefile.cpp -o corefile
ubuntu@ubuntu:~$ ./corefile
2
Give x and y as input.
```

c. **Finding Process id using pidof corefile program running.**



```
ubuntu@ubuntu:~$ pidof corefile
6983
ubuntu@ubuntu:~$ 
```

d. **Attaching gdb with a program running with above mentioned process id.**



```
ubuntu@ubuntu:~$ sudo gdb "/home/ubuntu/corefile" -p 6983
[sudo] password for ubuntu:
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 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 "aarch64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://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"...
Reading symbols from /home/ubuntu/corefile...
Attaching to program: /home/ubuntu/corefile, process 6983
Reading symbols from /lib/aarch64-linux-gnu/libstdc++.so.6...
(No debugging symbols found in /lib/aarch64-linux-gnu/libstdc++.so.6)
Reading symbols from /lib/aarch64-linux-gnu/libc.so.6...
Reading symbols from /usr/lib/debug/.build-id/3b/a44e06b9dc66aeeb2651db4dd015ffa
f6e0849.debug...
```

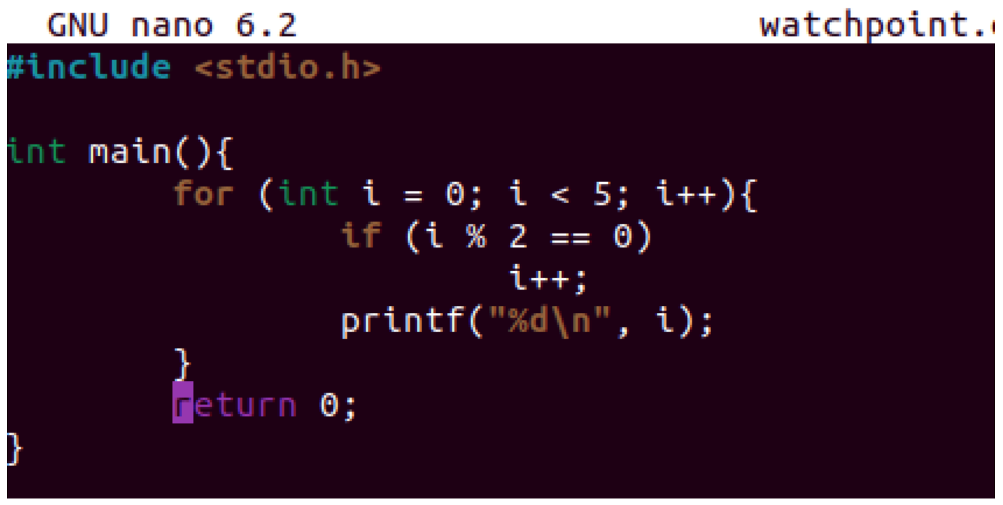
e. **Now we can do debugging in our normal manner.**

17) Some more advanced concepts based on your interest - for example 'watchpoint':

Watchpoints are a special kind of breakpoint which is bound to a variable rather than line number.

When the value of the variable being watched changes, the program is stopped.

Consider a simple program:



```
  GNU nano 6.2                               watchpoint.
#include <stdio.h>

int main(){
        for (int i = 0; i < 5; i++){
                if (i % 2 == 0)
                        i++;
                printf("%d\n", i);
        }
        return 0;
}
```

Lets see the watch point in action

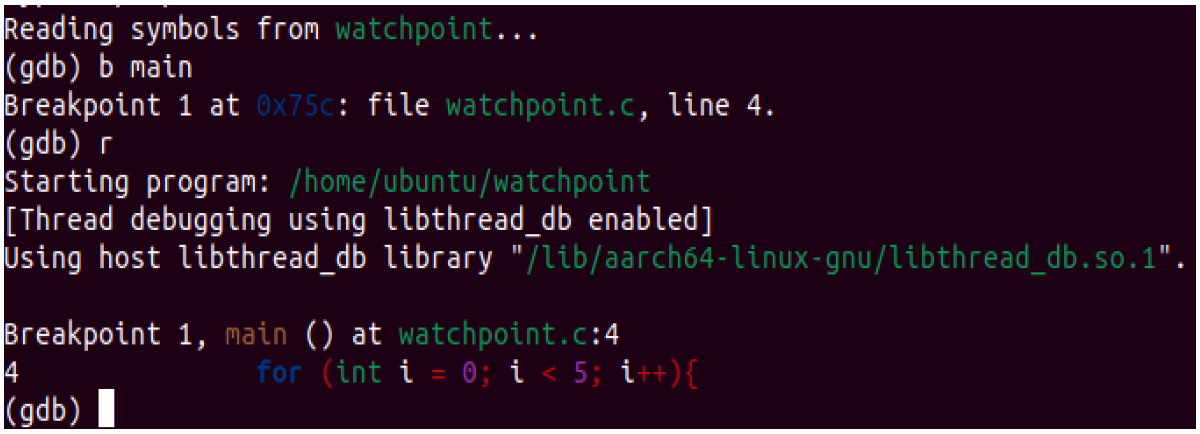
1. **compile the code: “gcc -g watchpoint.c -o watchpoint"**

2. **open the program with gdb: “gdb watchpoint”**

3. **add a breakpoint in main to stop the program just after the execution starts: “b main”**

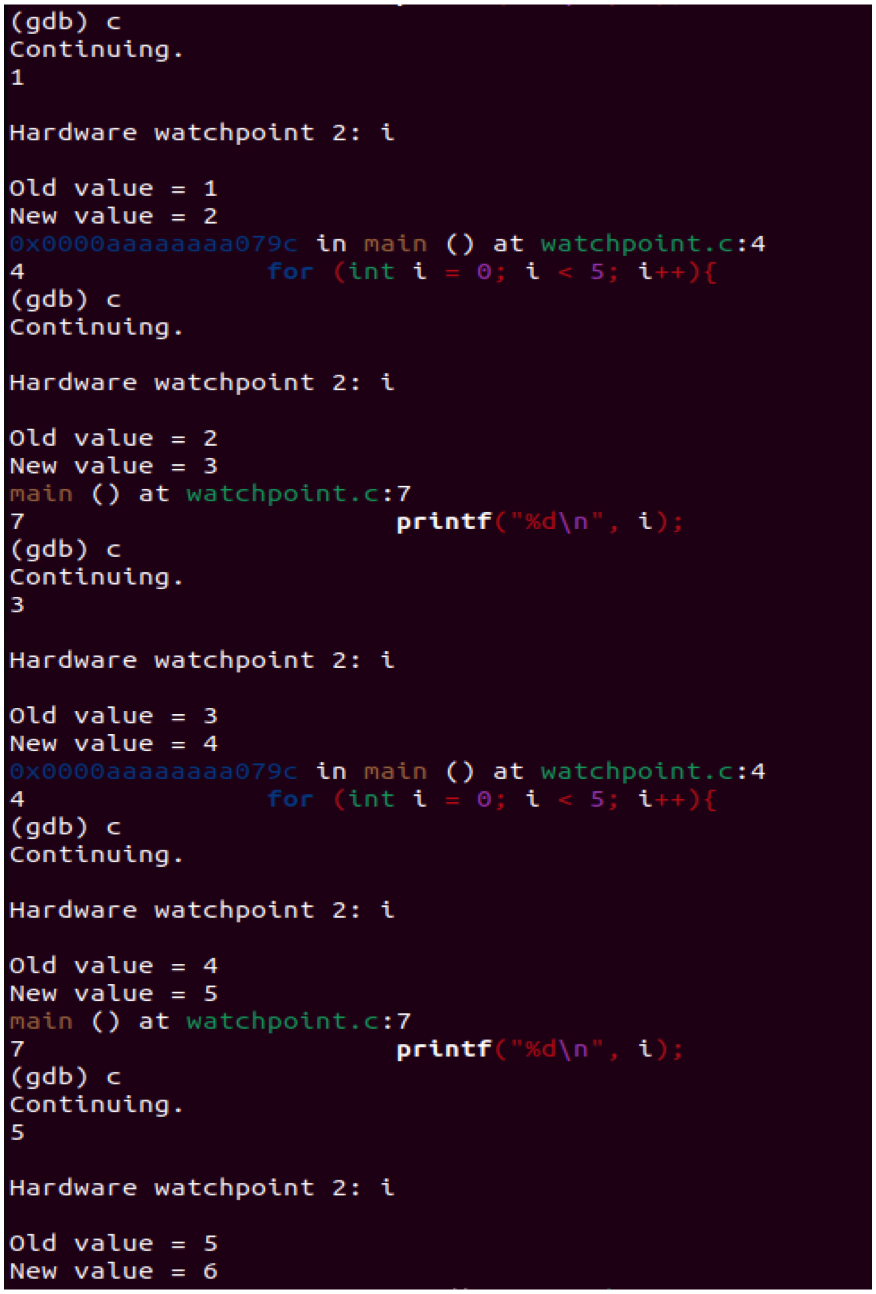
4. **run the program to start execution of program: “r”**

    a. **The program will stop just after the start of execution**


```
Reading symbols from watchpoint...
(gdb) b main
Breakpoint 1 at 0x75c: file watchpoint.c, line 4.
(gdb) r
Starting program: /home/ubuntu/watchpoint
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/aarch64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at watchpoint.c:4
4               for (int i = 0; i < 5; i++){
(gdb) 
```

5. add a watchpoint to i: “watch i”

6. Now keep running the program with “c”, we observe that the program stops everytime i changes


```
(gdb) c
Continuing.
1

Hardware watchpoint 2: i

Old value = 1
New value = 2
0x0000aaaaaaaa079c in main () at watchpoint.c:4
4               for (int i = 0; i < 5; i++){
(gdb) c
Continuing.

Hardware watchpoint 2: i

Old value = 2
New value = 3
main () at watchpoint.c:7
7                       printf("%d\n", i);
(gdb) c
Continuing.
3

Hardware watchpoint 2: i

Old value = 3
New value = 4
0x0000aaaaaaaa079c in main () at watchpoint.c:4
4               for (int i = 0; i < 5; i++){
(gdb) c
Continuing.

Hardware watchpoint 2: i

Old value = 4
New value = 5
main () at watchpoint.c:7
7                       printf("%d\n", i);
(gdb) c
Continuing.
5

Hardware watchpoint 2: i

Old value = 5
New value = 6
```