SOFTWARE ENGINEERING LAB
Assignment 2
TOPIC – Exploring GDB commands

Name	**Enrollment Number**
Ranveer Sahay	**2021CSB024**
Devans Soni	**2021CSB026**
M Vamsi Swarup	**2021CSB028**
Raksha Pahariya	**2021CSB029**
Anirban Debnath	**2021CSB030**

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

GNU Debugger (GDB) stands as a robust command-line utility utilized for debugging software applications within Linux and analogous Unix-like operating systems. Its primary function is to assist developers in identifying and rectifying errors within their code. This is achieved through a variety of functionalities, including the establishment of breakpoints, inspection of variables and memory, and the ability to navigate through code execution.

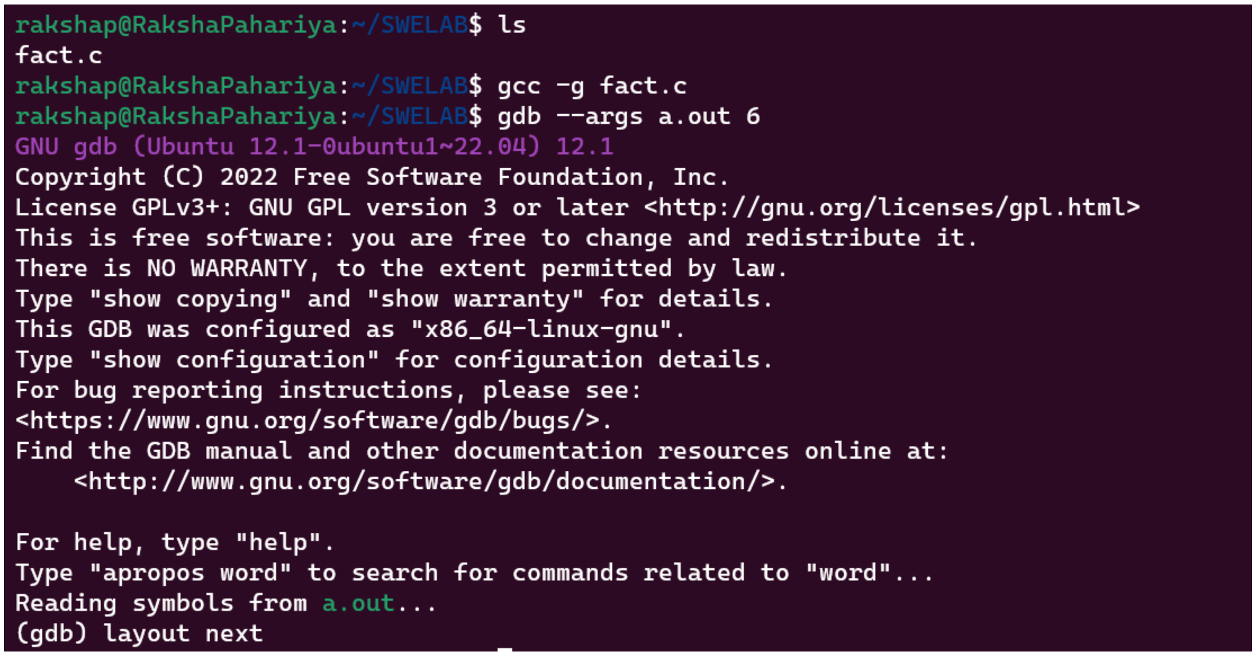
Typically, GDB operates alongside a compiler such as GCC, enabling the creation of an executable file embedded with debugging symbols. These symbols facilitate the loading of the program into the debugger. Once operational within GDB, developers gain access to an array of commands for scrutinizing and altering its execution. Such commands encompass the placement of breakpoints at specific code lines, step-by-step traversal of the program, examination of variable contents, and more.

Notably, GDB extends its support to a diverse spectrum of programming languages, encompassing C, C++, Python, Ada, and others. Moreover, it boasts formidable scripting capabilities, empowering developers to automate repetitive debugging tasks and tailor the debugger's functionality to align with their requirements.

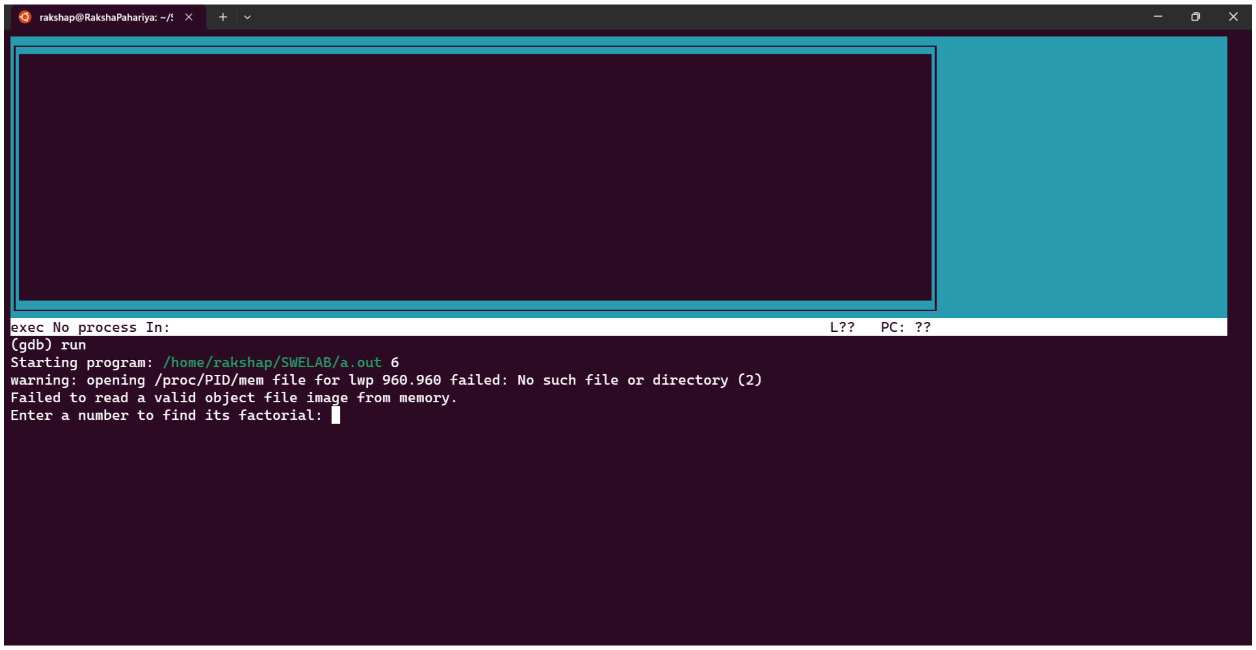
In summary, GDB emerges as an indispensable asset for software developers immersed in Linux and similar Unix-like environments. Its robust debugging toolkit, coupled with extensive language backing, positions it as an indispensable resource for efficiently and effectively debugging software applications.

1. Running a program

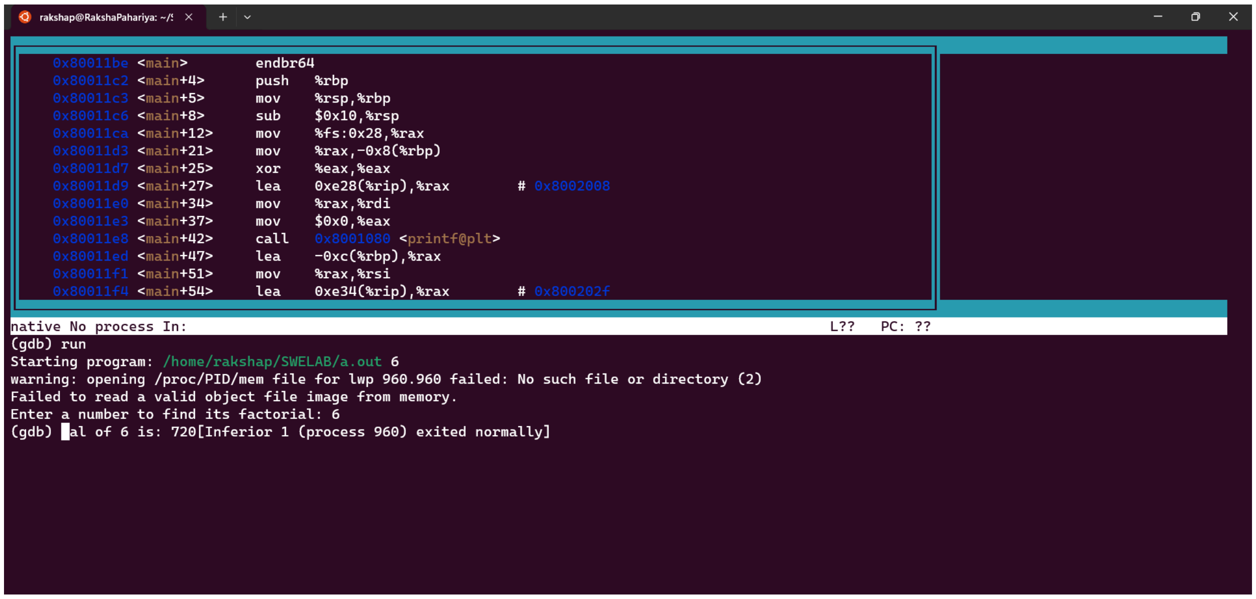
To prepare your program for debugging with gdb, you must compile it with the `-g` flag.
To start the program running, one can use the run command. If the program takes command-line arguments, they can be specified after the run command



```
rakshap@RakshaPahariya:~/SWELAB$ ls
fact.c
rakshap@RakshaPahariya:~/SWELAB$ gcc -g fact.c
rakshap@RakshaPahariya:~/SWELAB$ gdb --args a.out 6
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 "x86_64-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 a.out...
(gdb) layout next
```

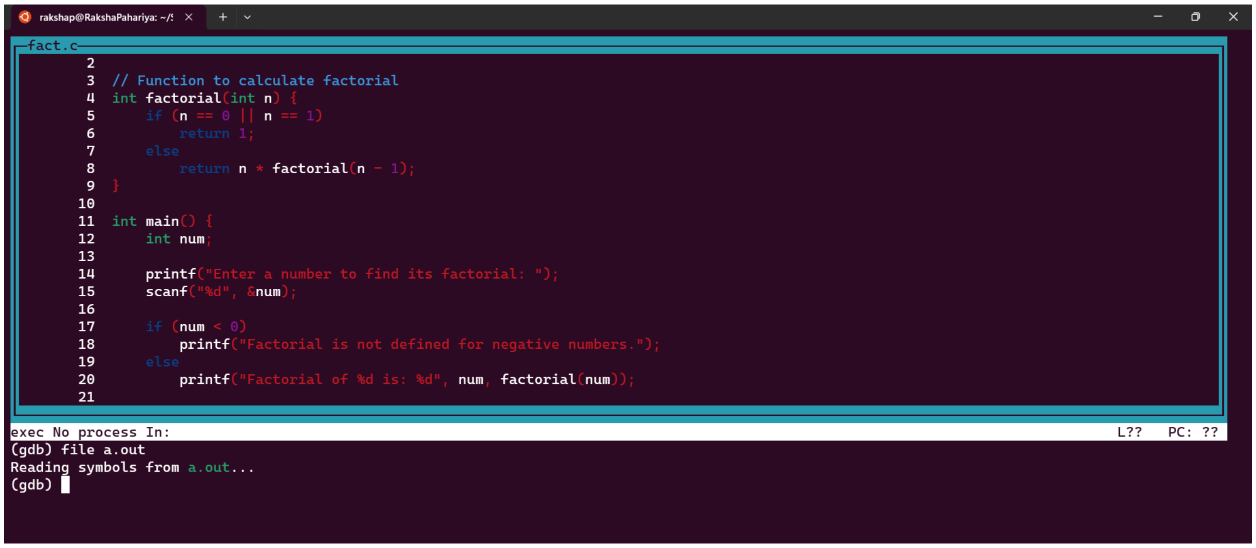


```
exec No process In:                                                       L??   PC: ??
(gdb) run
Starting program: /home/rakshap/SWELAB/a.out 6
warning: opening /proc/PID/mem file for lwp 960.960 failed: No such file or directory (2)
Failed to read a valid object file image from memory.
Enter a number to find its factorial: 
```


```
rakshap@RakshaPahariya: ~/!
    0x80011be <main>        endbr64
    0x80011c2 <main+4>      push   %rbp
    0x80011c3 <main+5>      mov    %rsp,%rbp
    0x80011c6 <main+8>      sub    $0x10,%rsp
    0x80011ca <main+12>     mov    %fs:0x28,%rax
    0x80011d3 <main+21>     mov    %rax,-0x8(%rbp)
    0x80011d7 <main+25>     xor    %eax,%eax
    0x80011d9 <main+27>     lea    0xe28(%rip),%rax        # 0x8002008
    0x80011e0 <main+34>     mov    %rax,%rdi
    0x80011e3 <main+37>     mov    $0x0,%eax
    0x80011e8 <main+42>     call   0x8001080 <printf@plt>
    0x80011ed <main+47>     lea    -0xc(%rbp),%rax
    0x80011f1 <main+51>     mov    %rax,%rsi
    0x80011f4 <main+54>     lea    0xe34(%rip),%rax        # 0x800202f
native No process In:                                                         L??   PC: ??
(gdb) run
Starting program: /home/rakshap/SWELAB/a.out 6
warning: opening /proc/PID/mem file for lwp 960.960 failed: No such file or directory (2)
Failed to read a valid object file image from memory.
Enter a number to find its factorial: 6
(gdb) █al of 6 is: 720[Inferior 1 (process 960) exited normally]
```

2. Loading symbol table

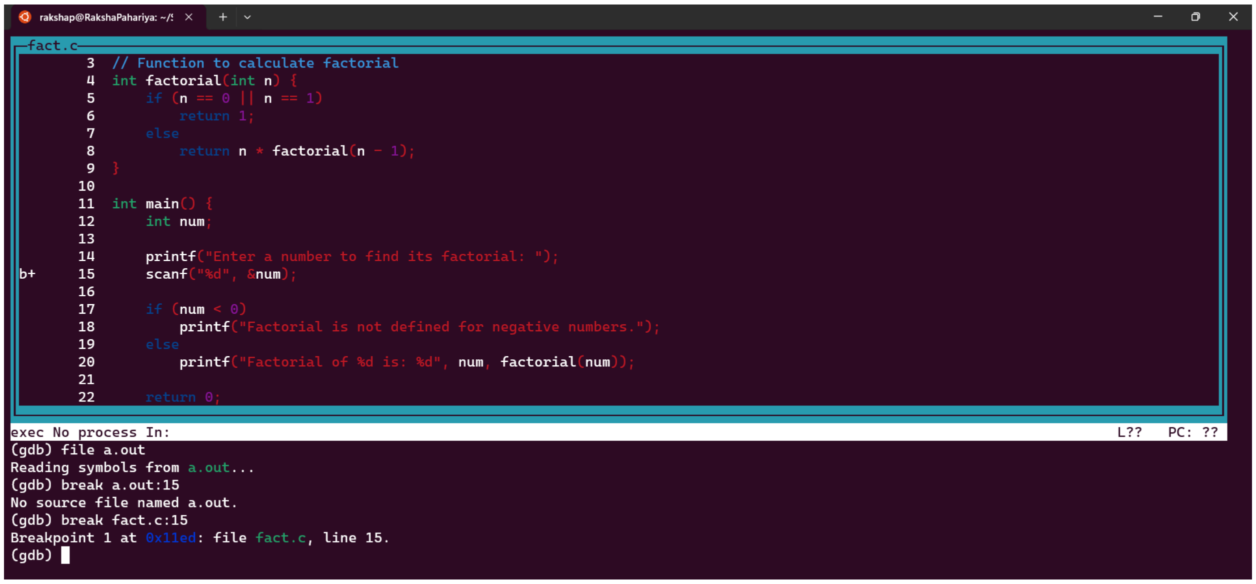
Once the program is compiled with debugging symbols, it can be loaded into GDB using the file


```
rakshap@RakshaPahariya: ~/!
fact.c

// Function to calculate factorial
int factorial(int n) {
    if (n == 0 || n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}

int main() {
    int num;

    printf("Enter a number to find its factorial: ");
    scanf("%d", &num);

    if (num < 0)
        printf("Factorial is not defined for negative numbers.");
    else
        printf("Factorial of %d is: %d", num, factorial(num));

exec No process In:                                                           L??   PC: ??
(gdb) file a.out
Reading symbols from a.out...
(gdb) █
```

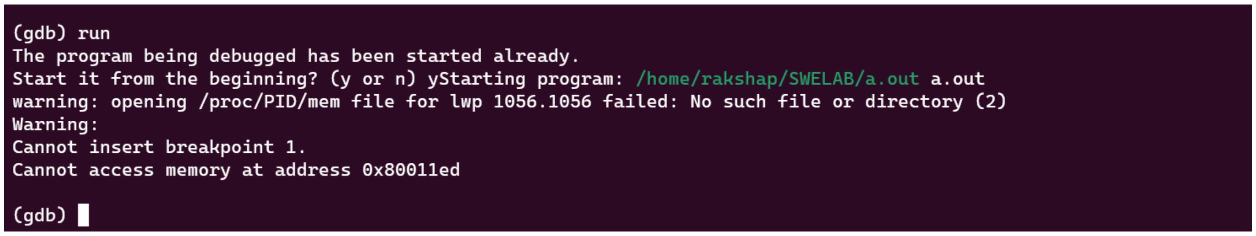
3. Setting a break-point

Normally, your program only stops when it exits. Breakpoints allow you to pause your program's execution wherever you want, be it at a function call or a particular line of code, and examine the program state.

Before you start your program running, you want to set up your breakpoints. The break command (shorthand: b) allows you to do so.



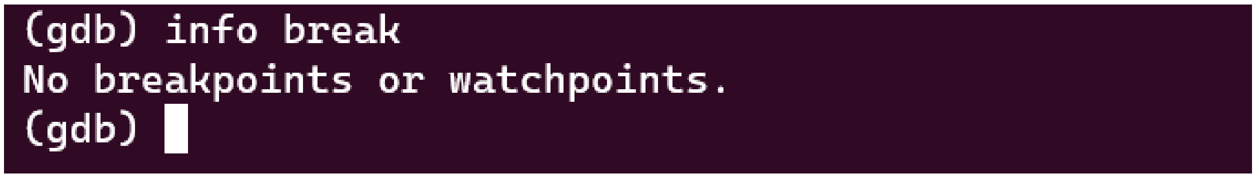
```
fact.c
         3  // Function to calculate factorial
         4  int factorial(int n) {
         5      if (n == 0 || n == 1)
         6          return 1;
         7      else
         8          return n * factorial(n - 1);
         9  }
        10
        11  int main() {
        12      int num;
        13
        14      printf("Enter a number to find its factorial: ");
b+      15      scanf("%d", &num);
        16
        17      if (num < 0)
        18          printf("Factorial is not defined for negative numbers.");
        19      else
        20          printf("Factorial of %d is: %d", num, factorial(num));
        21
        22      return 0;
exec No process In:                                                          L??   PC: ??
(gdb) file a.out
Reading symbols from a.out...
(gdb) break a.out:15
No source file named a.out.
(gdb) break fact.c:15
Breakpoint 1 at 0x11ed: file fact.c, line 15.
(gdb)
```



```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) yStarting program: /home/rakshap/SWELAB/a.out a.out
warning: opening /proc/PID/mem file for lwp 1056.1056 failed: No such file or directory (2)
Warning:
Cannot insert breakpoint 1.
Cannot access memory at address 0x80011ed

(gdb)
```

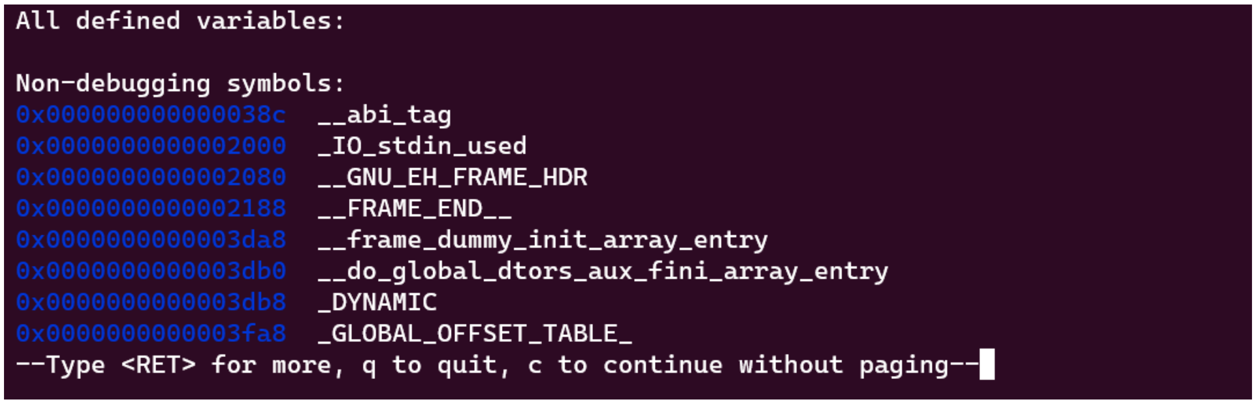
To delete the breakpoint numbered 2:
(gdb) delete 2

If you lose track of your breakpoints, or you want to see their numbers again, the info break command lets you know the breakpoint numbers:



```
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

4. Listing variables and examining their values


```
All defined variables:

Non-debugging symbols:
0x000000000000038c  __abi_tag
0x0000000000002000  _IO_stdin_used
0x0000000000002080  __GNU_EH_FRAME_HDR
0x0000000000002188  __FRAME_END__
0x0000000000003da8  __frame_dummy_init_array_entry
0x0000000000003db0  __do_global_dtors_aux_fini_array_entry
0x0000000000003db8  _DYNAMIC
0x0000000000003fa8  _GLOBAL_OFFSET_TABLE_
--Type <RET> for more, q to quit, c to continue without paging--
```

5. Printing content of an array or contiguous memory

In GDB, programmers have the capability to display the contents of an array or a contiguous block of memory while the program is running. This functionality proves invaluable for debugging tasks, enabling programmers to scrutinize memory contents and confirm their conformity to expected values.

To showcase the contents of an array in GDB, the `print` command is employed along with the array's name and the desired index range. This index range can be defined using either absolute indices or pointer arithmetic.

For instance, to reveal the contents of an array named `my_array` with 10 elements in GDB, the following command can be utilized:

```
(gdb) print my_array[0]@10
```

This command displays the first 10 elements of the array, commencing from index 0. In cases of multi-dimensional arrays, the index range for each dimension can be delineated, separated by commas.

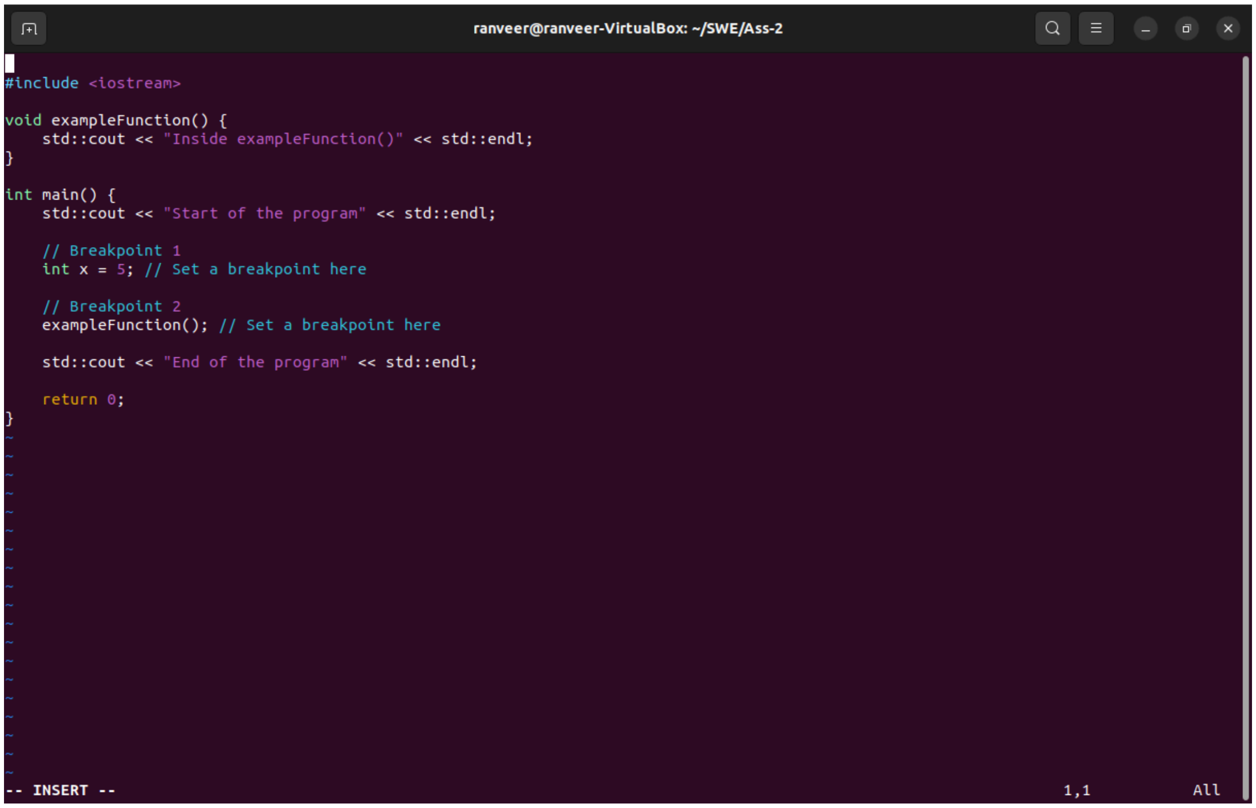
6. Printing function arguments

To display the arguments of a function in GDB, programmers can employ the "info args" command. This command provides a comprehensive list of all arguments transmitted to the ongoing function, along with their respective types and values.

For instance, to exhibit the arguments of the present function in GDB, one may utilize the following command:

(gdb) info args

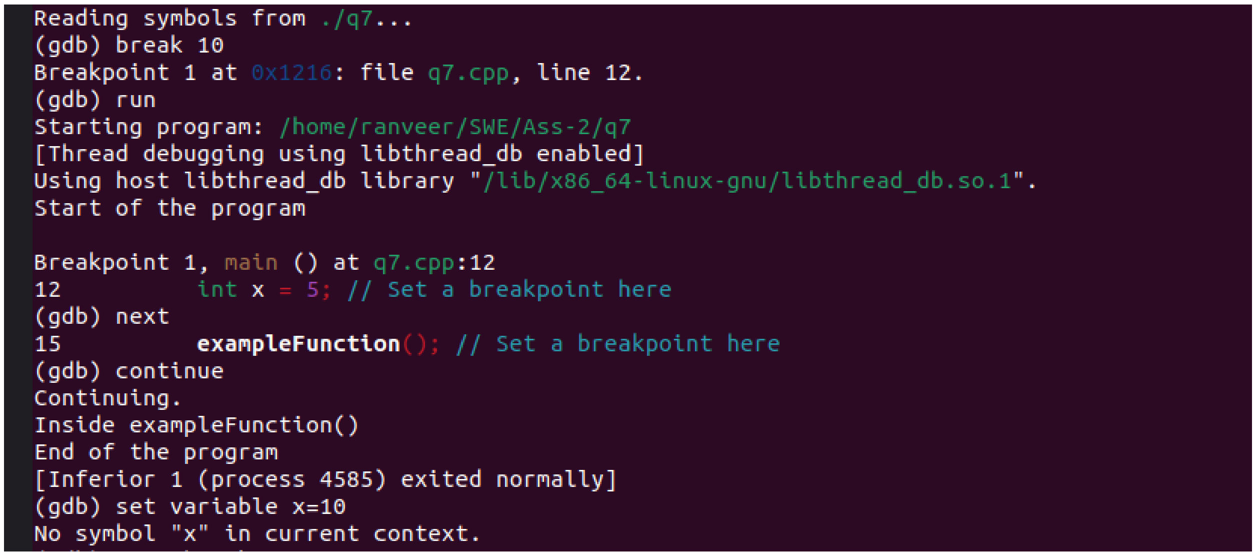
7. Next, Continue, Set command



ranveer@ranveer-VirtualBox: ~/SWE/Ass-2

```
#include <iostream>

void exampleFunction() {
    std::cout << "Inside exampleFunction()" << std::endl;
}

int main() {
    std::cout << "Start of the program" << std::endl;

    // Breakpoint 1
    int x = 5; // Set a breakpoint here

    // Breakpoint 2
    exampleFunction(); // Set a breakpoint here

    std::cout << "End of the program" << std::endl;

    return 0;
}
```

```
-- INSERT --                                                    1,1           All
```



```
Reading symbols from ./q7...
(gdb) break 10
Breakpoint 1 at 0x1216: file q7.cpp, line 12.
(gdb) run
Starting program: /home/ranveer/SWE/Ass-2/q7
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Start of the program

Breakpoint 1, main () at q7.cpp:12
12          int x = 5; // Set a breakpoint here
(gdb) next
15          exampleFunction(); // Set a breakpoint here
(gdb) continue
Continuing.
Inside exampleFunction()
End of the program
[Inferior 1 (process 4585) exited normally]
(gdb) set variable x=10
No symbol "x" in current context.
```

The `next` command in GDB facilitates executing the subsequent line of code within the program without delving into any function calls. This command proves beneficial for swiftly navigating through a program's execution flow while circumventing the intricacies of function calls. For instance, to execute the next line of code in GDB, one can invoke the following command:

```
next
```

On the other hand, the `continue` command serves the purpose of resuming program execution until encountering the subsequent breakpoint or program termination. It aids in restoring normal program flow after halting at a breakpoint. For instance, to resume program execution in GDB, the following command can be utilized:

```
continue
```

Furthermore, the `set` command in GDB offers the capability to alter variable values or debugger options during the debugging process. This functionality proves advantageous for experimenting with diverse scenarios without necessitating program re-execution. For instance, to assign a value of 10 to a variable named `my_var` in GDB, the ensuing command can be employed:

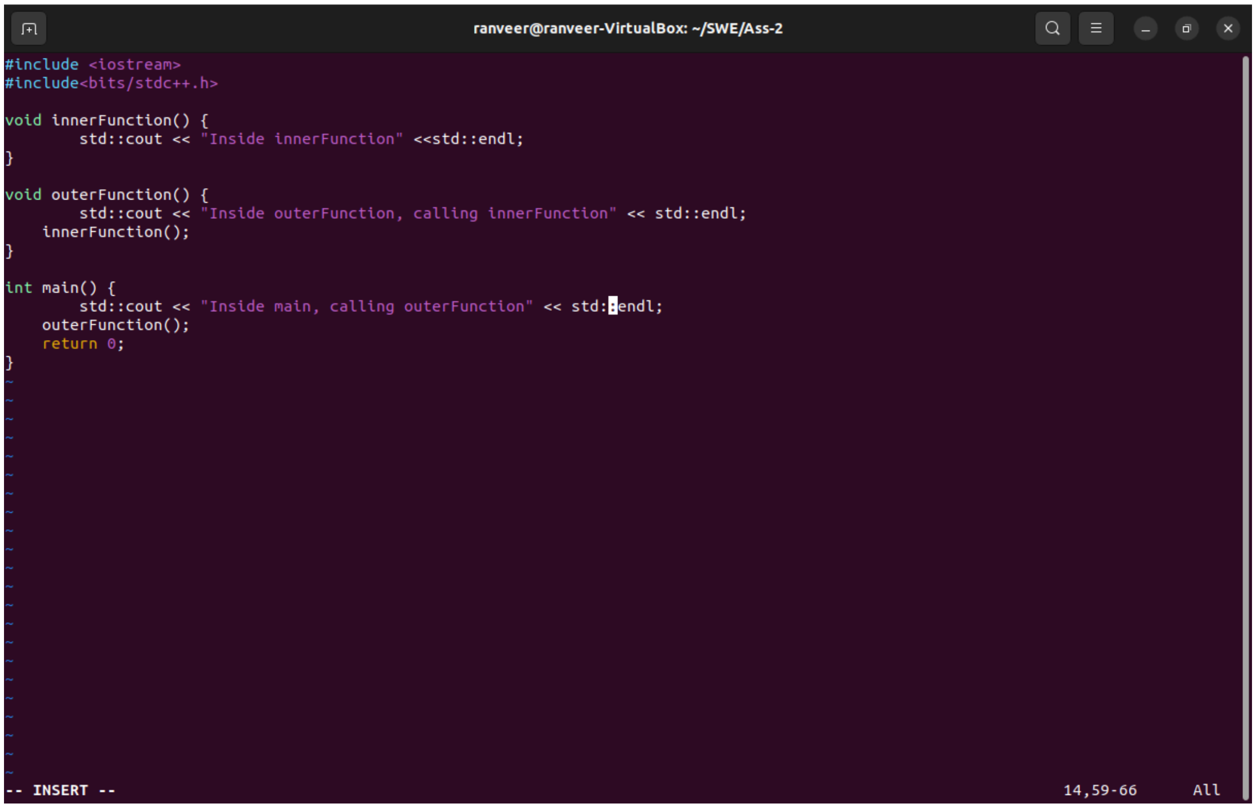
```
set variable my_var = 10
```

Moreover, the `set` command can be leveraged to modify the program's behavior by configuring various options. For instance, the `set print array` option dictates how GDB presents array values during debugging. To activate the `set print array` option in GDB, the subsequent command can be utilized:

```
set print array on
```

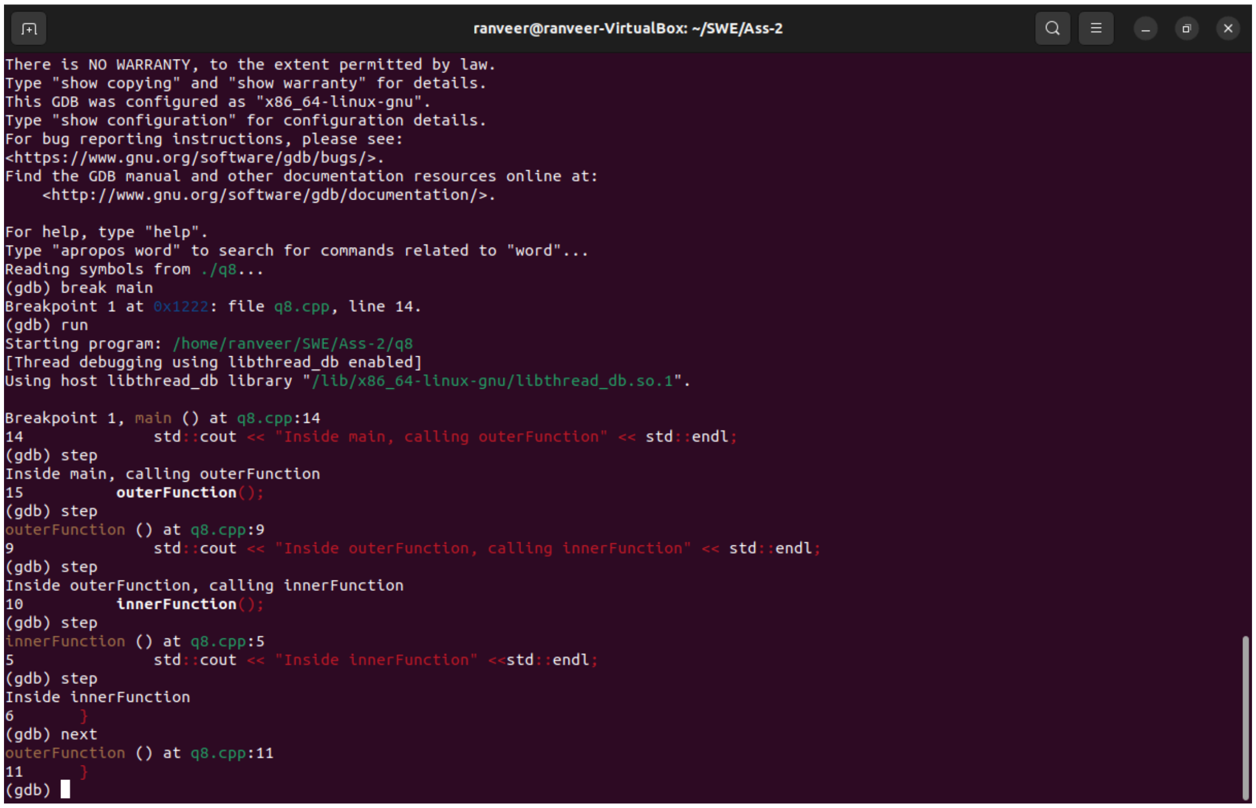
By leveraging the `set` command, programmers gain the flexibility to adjust program behavior and investigate different scenarios during the debugging phase. This capability empowers them to discern the intricacies of their program's behavior and diagnose errors more effectively.

8. Single stepping into function



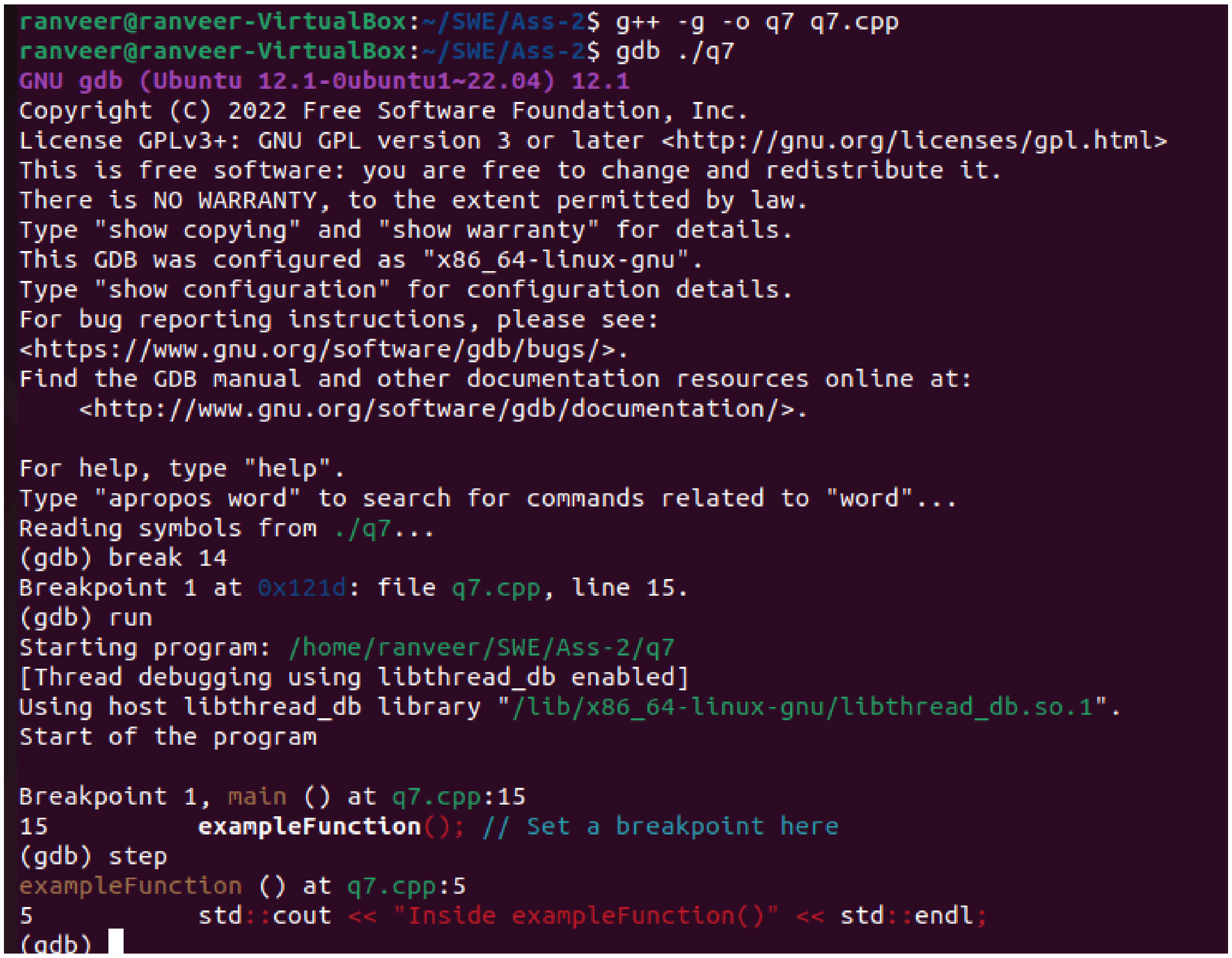
ranveer@ranveer-VirtualBox: ~/SWE/Ass-2

```
#include <iostream>
#include<bits/stdc++.h>

void innerFunction() {
        std::cout << "Inside innerFunction" <<std::endl;
}

void outerFunction() {
        std::cout << "Inside outerFunction, calling innerFunction" << std::endl;
    innerFunction();
}

int main() {
        std::cout << "Inside main, calling outerFunction" << std::endl;
    outerFunction();
    return 0;
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
-- INSERT --                                                                                       14,59-66      All
```



ranveer@ranveer-VirtualBox: ~/SWE/Ass-2

```
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<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 ./q8...
(gdb) break main
Breakpoint 1 at 0x1222: file q8.cpp, line 14.
(gdb) run
Starting program: /home/ranveer/SWE/Ass-2/q8
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at q8.cpp:14
14              std::cout << "Inside main, calling outerFunction" << std::endl;
(gdb) step
Inside main, calling outerFunction
15          outerFunction();
(gdb) step
outerFunction () at q8.cpp:9
9               std::cout << "Inside outerFunction, calling innerFunction" << std::endl;
(gdb) step
Inside outerFunction, calling innerFunction
10          innerFunction();
(gdb) step
innerFunction () at q8.cpp:5
5               std::cout << "Inside innerFunction" <<std::endl;
(gdb) step
Inside innerFunction
6       }
(gdb) next
outerFunction () at q8.cpp:11
11      }
(gdb) 
```



```
ranveer@ranveer-VirtualBox:~/SWE/Ass-2$ g++ -g -o q7 q7.cpp
ranveer@ranveer-VirtualBox:~/SWE/Ass-2$ gdb ./q7
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 "x86_64-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 ./q7...
(gdb) break 14
Breakpoint 1 at 0x121d: file q7.cpp, line 15.
(gdb) run
Starting program: /home/ranveer/SWE/Ass-2/q7
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Start of the program

Breakpoint 1, main () at q7.cpp:15
15          exampleFunction(); // Set a breakpoint here
(gdb) step
exampleFunction () at q7.cpp:5
5           std::cout << "Inside exampleFunction()" << std::endl;
(gdb)
```

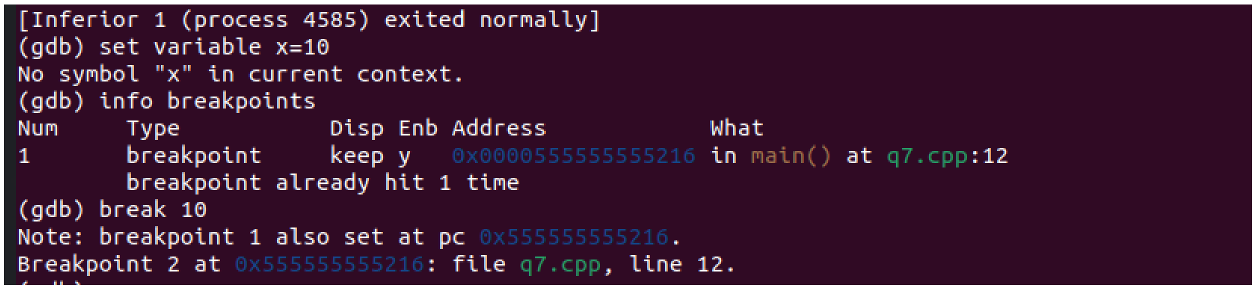
In GDB, stepping into a function call necessitates the programmer to initially establish a breakpoint within the function of interest. This can be accomplished by employing the "break" command, followed by specifying the name of the function.

Subsequently, after setting the breakpoint, the program can be initiated utilizing the "run" command. Upon reaching the designated breakpoint during program execution, the programmer can then utilize the "step" command to proceed into the function call, thereby scrutinizing its internal execution.

For instance, to delve into the function call at the current breakpoint in GDB, the following command sequence can be utilized:

(gdb) step

9. Listing all break point

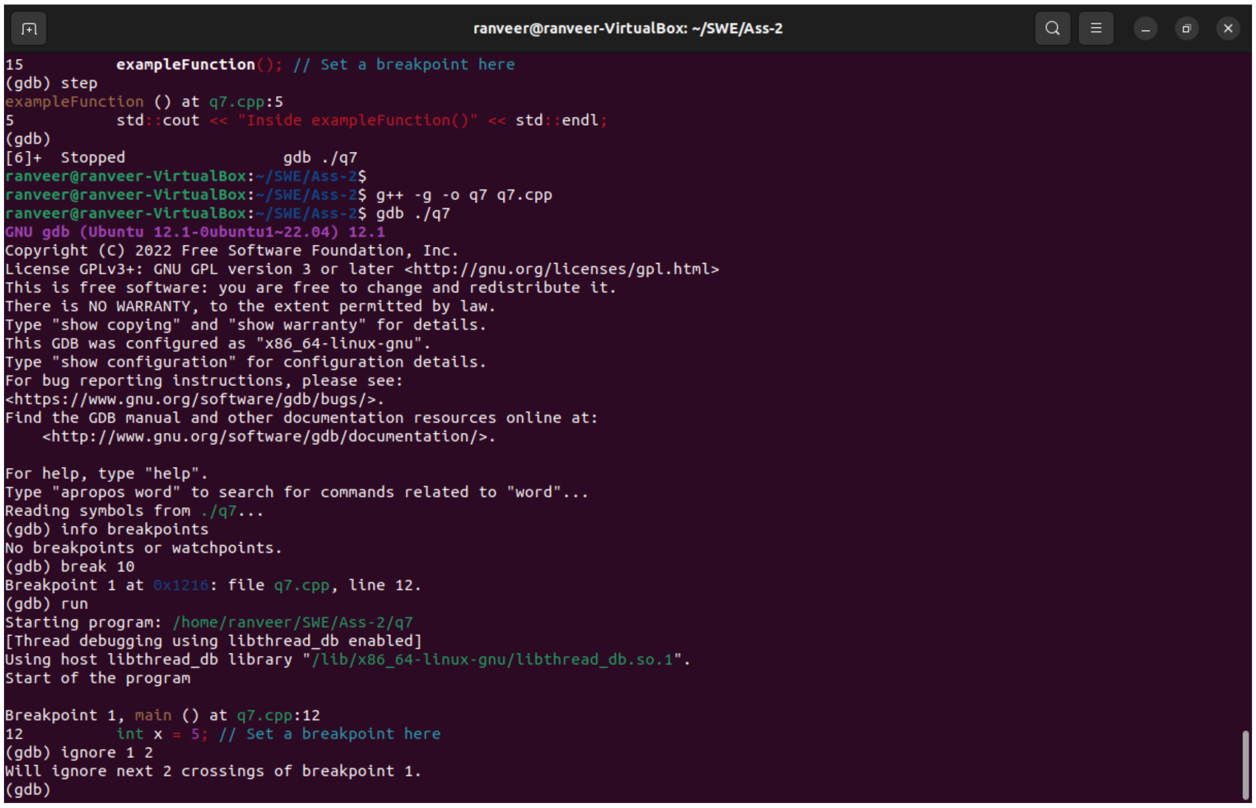


```
[Inferior 1 (process 4585) exited normally]
(gdb) set variable x=10
No symbol "x" in current context.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000555555555216 in main() at q7.cpp:12
        breakpoint already hit 1 time
(gdb) break 10
Note: breakpoint 1 also set at pc 0x555555555216.
Breakpoint 2 at 0x555555555216: file q7.cpp, line 12.
```

To list all the breakpoints we can use the command

(gdb) info break

10. Ignoring a break-point for N occurrence



ranveer@ranveer-VirtualBox: ~/SWE/Ass-2

```
15          exampleFunction(); // Set a breakpoint here
(gdb) step
exampleFunction () at q7.cpp:5
5           std::cout << "Inside exampleFunction()" << std::endl;
(gdb)
[6]+  Stopped                 gdb ./q7
ranveer@ranveer-VirtualBox:~/SWE/Ass-2$
ranveer@ranveer-VirtualBox:~/SWE/Ass-2$ g++ -g -o q7 q7.cpp
ranveer@ranveer-VirtualBox:~/SWE/Ass-2$ gdb ./q7
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 "x86_64-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 ./q7...
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb) break 10
Breakpoint 1 at 0x1216: file q7.cpp, line 12.
(gdb) run
Starting program: /home/ranveer/SWE/Ass-2/q7
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Start of the program

Breakpoint 1, main () at q7.cpp:12
12          int x = 5; // Set a breakpoint here
(gdb) ignore 1 2
Will ignore next 2 crossings of breakpoint 1.
(gdb)
```

To ignore a breakpoint in GDB, the programmer can use the following command:

(gdb) ignore [breakpoint number] [count]

Where [breakpoint number] is the number of the breakpoint to be ignored (as listed by the info break command) and [count] is the number of times the breakpoint should be ignored before stopping at it.

11. Enable/disable a break-point

In GDB, the commands "enable" and "disable" serve to activate or deactivate breakpoints, respectively. This functionality proves valuable in scenarios where a breakpoint is causing issues or when the programmer prefers to temporarily suspend a breakpoint without deleting it.

To deactivate a breakpoint in GDB, programmers can utilize the following command structure:

```
disable [breakpoint number]
```

Here, [breakpoint number] denotes the identifier of the breakpoint intended for deactivation, as enumerated by the "info break" command. For instance, to disable breakpoint number 2 in GDB, the subsequent command can be employed:

```
disable 2
```

This action temporarily suspends the breakpoint, thereby preventing program interruption at that specific location.

To reactivate a disabled breakpoint, programmers can utilize the "enable" command, specifying the same breakpoint number:

```
enable [breakpoint number]
```

For instance, to restore the functionality of breakpoint number 2 in GDB, the following command can be employed:

```
enable 2
```

This action restores the breakpoint's functionality, enabling the program to halt at the designated location as usual.

By employing the "enable" and "disable" commands in GDB, programmers attain enhanced control over their breakpoints, facilitating more efficient debugging processes.

12. Break condition and Command

In GDB, the "breakpoint" command facilitates the establishment of a breakpoint endowed with both a condition and a command. A condition comprises an expression evaluated each time the breakpoint is reached, and the breakpoint only halts program execution if the expression evaluates to true. Meanwhile, a command entails a sequence of GDB commands executed whenever the breakpoint is encountered.

To configure a breakpoint with a condition and a command in GDB, programmers can adhere to the ensuing syntax:

```
break [location] if [condition] commands [commands]
```

Here, [location] signifies the breakpoint's location, [condition] represents the condition expression, and [commands] constitutes the succession of GDB commands to be executed upon breakpoint encounter.

For instance, to establish a breakpoint at line 10 of the "my_function" function with a condition stipulating program interruption only if the variable "x" holds a value greater than 5, and to execute a command printing the value of the variable "y" each time the breakpoint is reached, the following command can be employed:

```
break my_function.c:10 if x > 5 commands print y
```

This action configures a breakpoint at line 10 of the "my_function" function, encompassing the designated condition and command. Whenever the breakpoint is hit and the condition holds true, program execution will pause, and the value of "y" will be printed.

13. Examining stack trace

In GDB, the "backtrace" or "bt" command serves to inspect the stack trace of the program, illustrating the sequence of function calls leading up to the current program point.

To scrutinize the stack trace in GDB, programmers can utilize the following command:

```
backtrace
```

Alternatively, the abbreviated version can be used:

```
bt
```

Executing this command will present the current stack trace of the program, showcasing the function calls in reverse chronological order, with the most recent calls listed first. For instance, when the program is halted at a breakpoint, invoking the "bt" command in GDB will reveal the stack trace up to that specific point.

Additionally, aside from delineating the function names, the stack trace also provides insights into the file names and line numbers where each function was invoked. This feature proves invaluable for bug tracking and comprehending the program's flow.

14. Examining stack trace for multi-threaded program

In a multi-threaded program, navigating the stack trace in GDB can be more intricate due to the simultaneous execution of multiple threads. To analyze the stack trace for a specific thread, GDB offers the "thread" command.

To explore the stack trace for a particular thread in GDB, programmers can execute the following command:

```
thread [thread number]
```

Here, [thread number] represents the identifier of the thread to be examined. For instance, to investigate the stack trace for thread number 3 in GDB, the following command can be used:

```
thread 3
```

This command switches the active thread to thread number 3 and showcases the corresponding stack trace.

Moreover, apart from scrutinizing the stack trace for a singular thread, GDB furnishes commands to inspect the status and stack traces of all threads within the program. The "info threads" command provides a synopsis of all threads, while the "thread apply" command facilitates the application of a command to all threads.

For instance, to unveil the stack traces of all threads in the program, the following command can be employed:

```
```

```
thread apply all bt
```

This command applies the "bt" command to all threads, revealing the stack trace for each individual thread.

By scrutinizing the stack traces of individual threads or all threads within a program, programmers can glean insights into the execution of a multi-threaded application and enhance their comprehension of how various threads interact with one another.

15. Core file debugging

In Linux, when a program crashes or encounters a fatal error, it often generates a core dump file. This file captures a snapshot of the program's memory state at the time of the crash, facilitating post-mortem debugging using GDB.

To debug a program using a core dump file in GDB, programmers can utilize the following command:

```
gdb [program] [core dump file]
```

Here, [program] refers to the name of the program that produced the core dump file, and [core dump file] denotes the path to the core dump file. For instance, to debug a program named my_program using a core dump file named core, one would execute:

```
gdb my_program core
```

This command initiates GDB and loads the core dump file, enabling programmers to examine the program's state at the time of the crash.

Once the core dump file is loaded into GDB, programmers can employ the standard GDB commands to inspect the program's memory, establish breakpoints, and traverse through the program's execution.

Utilizing a core dump file for post-mortem debugging empowers programmers to analyze the root cause of a program crash and pinpoint the underlying issues that precipitated the crash. This approach serves as a valuable tool for enhancing the quality and reliability of software.

16. Debugging of an already running program

In GDB, developers have the capability to attach to a running program and conduct debugging while it remains active. This feature proves invaluable for troubleshooting issues that manifest under specific conditions or for debugging lengthy programs that are challenging to replicate.

To attach GDB to a running program, developers can utilize the following command structure:

```
gdb [program] [process ID]
```

Here, [program] denotes the name of the program intended for debugging, while [process ID] represents the identifier of the running process. For instance, to attach GDB to an active instance of a program named my_program with a process ID of 1234, one would execute:

```
gdb my_program 1234
```

Once GDB establishes attachment to the running program, developers can employ the standard GDB commands to inspect the program's memory, establish breakpoints, and step through its execution flow.

However, it's crucial to acknowledge that attaching GDB to a running program carries risks, as it may potentially cause the program to crash or exhibit unpredictable behavior. Therefore, exercising caution during the debugging process is imperative, and having contingency plans in place is advisable in case of unexpected outcomes.

By leveraging GDB to debug a running program, developers can gain real-time insights into its behavior, enabling them to swiftly identify and rectify bugs as they arise.

17. Some more advance concepts based on your interest - for example 'watchpoint’

A watchpoint serves as a valuable debugging tool in GDB, allowing developers to track modifications to the value of a designated memory location or variable. When a watchpoint is established, GDB halts program execution and notifies the programmer whenever the monitored memory location or variable undergoes alteration.

To implement a watchpoint in GDB, developers utilize the following command structure:

```
```

```
watch [expression]
```

Here, [expression] denotes the memory location or variable that the developer intends to monitor. For instance, to establish a watchpoint on a variable named "count," the following command is employed:

```
watch count
```

Upon setting the watchpoint, GDB intervenes in the program's execution each time the value of the "count" variable changes.

Beyond setting a watchpoint on a specific variable or memory location, GDB supports conditional watchpoints, which activate solely when particular conditions are satisfied. For instance, developers can establish a watchpoint triggered exclusively when a variable is altered by a specific function.

Watchpoints function as potent debugging aids for pinpointing and troubleshooting issues related to memory location or variable modification. Through the utilization of watchpoints, developers can promptly identify instances and locations of variable changes, facilitating the diagnosis of underlying causes for these alterations.