

1 GNU Debugger(GDB)

What it is

GDB is a powerful debugging tool for C(and also for other languages like C++). It helps us to poke around inside our C programs while they are executing and also allows us to see what exactly happens when our program crashes.

Below is a C++ program:

```
1 #include<iostream>
2
3 using namespace std;
4
5 int main(void) {
6     int i=10, j;
7
8     j=i*10;
9     i+=20;
10
11     cout<<"Hello world\n";
12
13     return 0;
14 }
```

We will compile the program:

```
$ g++ -g main1.cpp -o main1
```

The `-g` flag is used to produce debugging information in the operating system's native format. GDB can work with this debugging information.

2 Using GDB

Running a program in GDB

To run a program in GDB, use the `gdb` command and the name of the program.

```
$ gdb .\main1.exe
```

When this command will be executed then we will get the following output:

```
PS D:\QuickGDB> gdb .\main1.exe
GNU gdb (GDB) 7.6.1
Copyright (C) 2013 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 "mingw32".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from D:\QuickGDB\main1.exe...done.
(gdb) █
```

Figure 1: Output of `gdb .\main.exe`

We can see that a prompt appears at the bottom(as `(gdb)`). We can type in commands in that prompt.

2.1 The *run* command

We will use the **run** command to run our program(**main1.cpp**).

This will give us the following output:

```
(gdb) run
Starting program: D:\QuickGDB/./main1.exe
[New Thread 26036.0x34d0]
[New Thread 26036.0x2df0]
[New Thread 26036.0x569c]
[New Thread 26036.0x5ad0]
Hello world
[Inferior 1 (process 26036) exited normally]
(gdb) █
```

Figure 2: Output of the **run** command

We can see that "Hello world" has been printed.

2.2 The *break* command

This command is used to set breakpoints in our program. Breakpoints are specific points in our code where the debugger will temporarily halt program execution. This allows us to inspect the program's state.

This is how we use the *break* command, just use the *break* command along with the line number, a function name or a *file:line* combination where we want to set the breakpoint.

```
(gdb) break main
```

Here, we are providing the function name **main** as the breakpoint.

We get the following output:

```
(gdb) break main
Breakpoint 1 at 0x40146e: file main1.cpp, line 6.
(gdb) █
```

Figure 3: Setting the breakpoint at the first line of main's body

Now, we will again use the **run** command.

So, after using the **run** command, this is the output:

```
(gdb) run
Starting program: D:\QuickGDB/.\main1.exe
[New Thread 11204.0x5080]
[New Thread 11204.0x5630]
[New Thread 11204.0x364]
[New Thread 11204.0x3360]

Breakpoint 1, main () at main1.cpp:6
6          int i=10, j;
(gdb) █
```

Figure 4: Execution stops at the first line of main's body

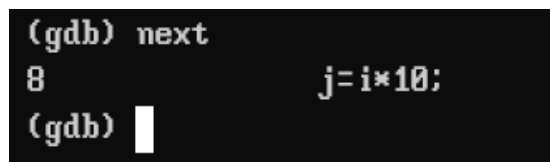
2.3 The *next* command

This command is used to execute the program until the next source line in the current function is reached. If the current line contains a function call, then the **next** command will execute the entire function and stop at the first source line after the function call returns.

Now, if we type next:

```
(gdb) next
```

This, basically, will print the next line after executing the 6th (this 6th line is where the last execution stopped) and will stop after printing this line (the line after the 6th line). Here is the output:

A screenshot of a GDB terminal window with a black background and white text. The first line shows the command '(gdb) next'. The second line shows the result '8' followed by a large amount of white space and then the source code line 'j=i*10;'. The third line shows the prompt '(gdb) ' followed by a white cursor bar.

```
(gdb) next
8                                     j=i*10;
(gdb) |
```

Figure 5: Output of **next** command

Here, we can see that the next source line after line 6 is 8. So, **next** stops at line 8 after executing line 6.

2.4 The *list* command

This command is used to display the source code around the current line of execution.

- If a function name is provided as an argument, it shows the source code for that function.
- If no argument is provided, it shows the source code around the current program counter.

We can also provide line numbers or ranges as arguments to the **list** command to view specific parts of the code. Example:

```
(gdb) list 20
```

This command shows the source code starting from line 20.

```
(gdb) list 30, 40
```

This command shows the source code from line 30 to line 40.

```
(gdb) list function.c:50, function.c:60
```

This command shows the source code from line 50 to line 60 in **function.c**.

```
(gdb) list
3      using namespace std;
4
5      int main(void){
6          int i=10, j;
7
8          j=i*10;
9          i+=20;
10
11          cout<<"Hello world\n";
12
(gdb) 
```

Figure 6: Output of **list** command

We can see that the **list** command shows us the lines before and the lines after line 8(which was the line till where the execution last stopped).

2.5 The *print* command

This command is used to evaluate and print the value of expressions or variables during program debugging.

Now, we have a variable named **j** in our program.

If we write:

```
(gdb) print j
```

this will basically print 0 because when we used **next** command it basically executed the line **int i=10, j;** in the program and it stopped at the line **j=i*10;** that means this line is not executed and therefore *j* is not initialized yet.

But:

```
(gdb) print i
```

This will print 10. Here is the output:

```
(gdb) print i
$2 = 10
```

Figure 7: Output of (gdb) print i

Now we use the **next** command again which gives us the following output:

```
(gdb) next
9          i+=20;
(gdb) print j
$4 = 100
```

Figure 8: The **next** command

Here, we executed line 8 and after executing this line **next** stops at line 9(i.e this is the next line to be executed). Notice that after executing line 8, we used the **print** command to print the value of *j* which is 100 after the multiplication operator.

The **print** command can also work with the pointer dereferencing operator(*). Say, we have a pointer in our program and we want to see to which address this pointer is pointing to, then we just need to write:

```
(gdb) print *ptr
```

We can also look at the memory address of a variable using the **print** command as shown below:

```
(gdb) print &j
```

This prints the memory address of *j*.

3 More

Consider the following program:

```
1  #include "blackBox.h"
2
3  using namespace std;
4
5  void crash(int *i) {
6      *i=1;
7  }
8
9  void f(int *i) {
10     int *j=i;
11
12     sophisticated(j);
13     j=complicated(j);
14
15     crash(j);
16 }
17
18 int main(void) {
19     int i;
20
21     f(&i);
22
23     return 0;
24 }
```

Here are the function definitions of **sophisticated()** and **complicated()**:

```
void sophisticated(int *j) {
    *j=10;
}

int* complicated(int *j) {
    return NULL;
}
```


So, this program throws a **Segmentation fault**:

```
priyanuj@Grafter:/mnt/d/QuickGDB$ g++ -g main2.cpp -o main2
priyanuj@Grafter:/mnt/d/QuickGDB$ ./main2
Segmentation fault
priyanuj@Grafter:/mnt/d/QuickGDB$
```

Figure 9: Our beloved **Segmentation fault**

I had to switch to WSL2 as running this program on Windows wasn't showing the SegFault!

3.1 Debugging the program

Start gdb:

```
$ gdb ./main2
```

Set a breakpoint, for me, I followed the tutorial and provided the breakpoint as follows:

```
(gdb) break crash
```

Now, we run our program:

```
(gdb) run
```

```
(gdb) break crash
Breakpoint 1 at 0x11c1: file main2.cpp, line 6.
(gdb) run
Starting program: /mnt/d/QuickGDB/main2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, crash (i=0x0) at main2.cpp:6
6          *i=1;
(gdb)
```

Figure 10: Notice **i**'s value

So, our execution stops at line 6 i.e ***i=1;**. This line is not executed. Notice the value of **i**. **i** is a pointer to an integer and the address in it is **0x0** which indicates that **i** stores the **NULL** value.

3.2 The *up* and *down* commands

- *up* → used to move up the call stack to the calling function.
- *down* → used to move down the call stack to the called function.

Here is what *up* gives us:

```
(gdb) up
#1  0x000055555555520e in f (i=0x7fffffffed4) at main2.cpp:15
15          crash(j);
(gdb) up
#2  0x0000555555555238 in main () at main2.cpp:21
21          f(&i);
(gdb) █
```

Figure 11: The *up* command

It is basically showing us the functions which are calling **crash()**, **f()** respectively.

Here is what *down* gives us:

```
(gdb) down
#1  0x000055555555520e in f (i=0x7fffffffed4) at main2.cpp:15
15          crash(j);
(gdb) down
#0  crash (i=0x0) at main2.cpp:6
6          *i=1;
(gdb) █
```

Figure 12: The *down* command

We can see that *down* is basically showing us the lines where **crash()** and **f()** are called.

3.3 The *display* command

This command is used to print the value of an expression each time the program stops. It's particularly useful for monitoring the value of variables or expressions as the program executes, providing a way to track changes in real-time.

```
(gdb) break f
Breakpoint 2 at 0x555555551de: file main2.cpp, line 10.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /mnt/d/QuickGDB/main2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 2, f (i=0x7fffffffe0d4) at main2.cpp:10
10      int *j=i;
(gdb) display j
1: j = (int *) 0x7ffff7ea67ea <std::basic_ios<wchar_t, std::char_traits<wchar_t> >::_M_cache_locale(std::locale const&)+90>
```

Figure 13: Setting a second breakpoint and running the program again. The *display* command

Notice, that program stops at line 10 in the function *f()* which is: `int *j=i;`. Notice the address stored in *j* which is displayed when we use the *display* command (*j* stores some random address).

```
(gdb) next
12      sophisticated(j);
1: j = (int *) 0x7fffffffe0d4
(gdb)
```

Figure 14: *j* now stores the address of *i*

Since, we used the *next* command, line 10 got executed and the address of *i* got stored in *j* as can be seen from figure 14.

Now, we will again run the **next** command and execute line 12 which is **sophisticated(j);**.

```
(gdb) next
13          j=sophisticated(j);
1: j = (int *) 0x7fffffffed4
(gdb) █
```

Figure 15: **sophisticated(j);** got executed

So, after executing line 12, the execution stops at line 13 which is **j=sophisticated(j);**. We can see that the address stored in **j** hasn't changed.

So, we run the **next** command again:

```
(gdb) next
15          crash(j);
1: j = (int *) 0x0
(gdb) █
```

Figure 16: **j=sophisticated(j);** got executed

Now, we can see that line 13 also got executed and execution stops at line 15. Notice that now the address stored in **j** has changed to **0x0** which means **NULL**.

3.4 The *undisplay* command

This is just the opposite of *display* command.

```
1: j = (int *) 0x0
(gdb) undisplay j
(gdb) up
#1  0x00005555555555238 in main () at main2.cpp:21
21      f(&i);
(gdb) next

Breakpoint 1, crash (i=0x0) at main2.cpp:6
6      *i=1;
(gdb) 
```

Figure 17: The *undisplay* command

Notice that the value of **j** stopped displaying. Also, an important note: if the **undisplay** command is throwing a warning on this syntax:

```
(gdb) undisplay j
```

In figure 17(at the very top), we can see **1: j=(int*) 0x0**. That 1 on the left indicates the *id* of the variable **j**. So, instead of using *undisplay* with the variable name, we can use the command with the variable's id:

```
(gdb) undisplay 1
```

3.5 The *backtrace* command

When we run this command in gdb, it prints a list of function calls, along with the corresponding stack frames, showing the current execution context. It includes function names, line numbers and source file information.

```
(gdb) backtrace
#0 crash (i=0x0) at main2.cpp:6
#1 0x000055555555520e in f (i=0x7fffffffe0d4) at main2.cpp:15
#2 0x0000555555555238 in main () at main2.cpp:21
(gdb) █
```

Figure 18: The *backtrace* command

3.6 More commands

Consider the following program:

```
1  #include<iostream>
2
3  using namespace std;
4
5  int factorial(int n) {
6      if(n==0)
7          return 1;
8
9      return n*factorial(n-1);
10 }
11
12 int main(void) {
13     int num, result;
14     cout<<"Enter an integer: ";
15     scanf("%d", &num);
16
17     if(num<0) {
18         cout<<"Number is negative!\n";
19     }
20
21     else {
22         result=factorial(num);
23         cout<<num<<"! = "<<result<<"\n";
24     }
25
26     return 0;
27 }
```

We set a breakpoint at the function **factorial**.

```
(gdb) break factorial
```

We will run the program:

```
(gdb) run
```

Here is an the image showing what is done:

```
(gdb) break factorial
Breakpoint 1 at 0x11f8: file main3.cpp, line 6.
(gdb) run
Starting program: /mnt/d/QuickGDB/main3
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Enter an integer: 4

Breakpoint 1, factorial (n=4) at main3.cpp:6
6         if(n==0)
(gdb) n
9         return n*factorial(n-1);
(gdb) |
```

Figure 19: Debugging a recursive program in gdb

n here is the abbreviated form of the **next** command.

Now, if we use the **next** command again, then it won't step into the function call at line 9(as shown in figure 19). It will just step over it. We want to step into this function call. For this we have the **step** command.

3.7 The *step* command

This command will go inside a function(unlike **next**) and execute that function line by line.

```
(gdb) step

Breakpoint 1, factorial (n=3) at main3.cpp:6
6         if(n==0)
(gdb) |
```

Figure 20: The **step** command

Notice that the value of **n** is 3. That means **step** command entered into the function call. But this process can get tedious with long recursive functions. We can't just type **step** everytime for executing each line in the function?

We can make use of the breakpoint. We can tell gdb to run as many lines as it can until it reaches the next breakpoint.

3.8 The `continue` command

This command is used to resume the execution of the program until the next breakpoint is encountered or until the program completes its execution. It allows the program to run continuously without stopping at every line or function call. It's quite similar to the `run` except that `continue` runs the program from wherever the execution stopped previously.

```
(gdb) continue
Continuing.

Breakpoint 1, factorial (n=2) at main3.cpp:6
6          if(n==0)
(gdb) █
```

Figure 21: The `continue` command

We can see that after using the `continue` command, the execution stopped right at the next breakpoint. Also, notice that the value of `n` is now 2. `continue` command is abbreviated as `c`. So, we go on continuing with the `continue` command until we reach the breakpoint where `n` is 0.

3.9 The *finish* command

This command is used to continue execution of the program until the current function is finished and control returns to the calling function. It allows us to finish the execution of the current function and stop at the first instruction of the calling function in the call stack.

Now, from the last breakpoint (where **n** became 0), we use the **finish** command.

```
(gdb) c
Continuing.

Breakpoint 1, factorial (n=0) at main3.cpp:6
6          if(n==0)
(gdb) finish
Run till exit from #0  factorial (n=0) at main3.cpp:6
factorial (n=1) at main3.cpp:9
9          return n*factorial(n-1);
Value returned is $1 = 1
(gdb)
Run till exit from #0  factorial (n=1) at main3.cpp:9
factorial (n=2) at main3.cpp:9
9          return n*factorial(n-1);
Value returned is $2 = 1
(gdb)
Run till exit from #0  factorial (n=2) at main3.cpp:9
factorial (n=3) at main3.cpp:9
9          return n*factorial(n-1);
Value returned is $3 = 2
(gdb)
Run till exit from #0  factorial (n=3) at main3.cpp:9
factorial (n=4) at main3.cpp:9
9          return n*factorial(n-1);
Value returned is $4 = 6
(gdb)
Run till exit from #0  factorial (n=4) at main3.cpp:9
0x0000555555555293 in main () at main3.cpp:22
22          result=factorial(num);
Value returned is $5 = 24
```

Figure 22: The *finish* command

Notice the line **Run till exit from #0**. The **#0** is basically the stack frame number. In GDB, the stack frames are numbered. The **#0** is the currently executing function, **#1** is the calling function and so on. Basically **#0** is the top of the stack.

Also, notice the **Value returned is** statement.

Finally, we reach **result=factorial(n);** in the **main()** function where the result(the factorial of 4) is stored.

```
(gdb) bt
#0  factorial (n=1) at main3.cpp:6
#1  0x0000555555555212 in factorial (n=2) at main3.cpp:9
#2  0x0000555555555212 in factorial (n=3) at main3.cpp:9
#3  0x0000555555555212 in factorial (n=4) at main3.cpp:9
#4  0x0000555555555293 in main () at main3.cpp:22
(gdb)
```

Figure 23: Using the *backtrace* command to view the stack

From figure 23, we can see the call stack where the calls of **factorial()** are stored. We can see that **factorial()** was first called in the **main()** function at line 22. Next, it was again called in line 9(within its definition) all the way till stack frame #1. Then in stack frame #0 **factorial()** was called again with **n=1** and the execution stopped at the breakpoint.

3.10 The *watch* command

This command is used to set a watchpoint on a variable or an expression. A watchpoint is a debugging feature that causes the program to stop execution whenever the specified variable or expression is read or modified.

Basic syntax:

```
watch EXPRESSION
```

- EXPRESSION → Any valid C/C++ expression, including variable names, array elements, or more complex expressions.

Consider the following program:

```
1 #include<iostream>
2
3 int main(void) {
4     int x=5;
5     int y=10;
6     int z=x+y;
7
8     printf("Initial values:\nx = %d\ny = %d\nz = %d\n", x, y, z);
9
10    x=20;
11    y=x*4;
12    z=x+10;
13
14    printf("Updated values:\nx = %d\ny = %d\nz = %d\n", x, y, z);
15    return 0;
16 }
```

Let's say we want to keep track of the variable `x`. We will use the *watch* command as:

```
(gdb) watch x
```

```
Breakpoint 1, main () at main4.cpp:4
4      int x=5;
(gdb) n
5      int y=10;
(gdb) p x
$1 = 5
(gdb) watch x
Hardware watchpoint 2: x
```

Figure 24: Setting a watchpoint with the *watch* command

Here we have created a watchpoint for the variable `x`.

```
(gdb) c
Continuing.
Initial values:
x = 5
y = 10
z = 15

Hardware watchpoint 2: x

Old value = 5
New value = 20
main () at main4.cpp:11
11      y=x*4;
(gdb)
```

Figure 25: Caught modification

As we continue executing the program, **watch** command lets us know the modified value of `x` by showing the old value and the new value. We can also see the modification occurred inside the `main()` function.

3.11 The *info* command

This command provides information about various aspects of the program being debugged.

To get information about breakpoints, we can use the following command:

```
(gdb) info breakpoints
```

```
(gdb) info breakpoints
Num      Type           Disp Enb Address            What
1        breakpoint     keep y   0x000055555555195 in main() at main4.cpp:4
          breakpoint already hit 1 time
2        hw watchpoint  keep y                   x
          breakpoint already hit 1 time
(gdb) █
```

Figure 26: *info* command

Notice that each breakpoint has an **id** associated with it (look at the **Num** column in figure 26).

3.12 The *delete* command

This command is used to delete breakpoints or watchpoints that have been set during the debugging session. It allows us to remove specific breakpoints, all breakpoints or even breakpoints at a specific location.

To delete a breakpoint, we use the breakpoint's id along with the **delete** command.

```
(gdb) delete 1
```

```
(gdb) delete 2
(gdb) info breakpoints
Num      Type             Disp Enb Address                What
1        breakpoint      keep y   0x0000555555555195 in main() at main4.cpp:4
          breakpoint already hit 1 time
(gdb) █
```

Figure 27: Deleting a breakpoint

To confirm that a breakpoint has been removed/deleted, we use the **info breakpoints** command to check it. We deleted our watchpoint (figure 27).

To delete all breakpoints and watchpoints, just use the **delete** command alone.

```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb) █
```

Figure 28: *delete* command alone

3.13 The *whatis* command

This command is used to display the data type of a symbol, variable, or expression in the program being debugged. It provides information about the type of the specified identifier without evaluating or executing any code.

Consider the program in the **first** section.

```
(gdb) break main
Breakpoint 1 at 0x1195: file main1.cpp, line 6.
(gdb) run
Starting program: /mnt/d/QuickGDB/main1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at main1.cpp:6
6          int i=10, j;
(gdb) n
8          j=i*10;
(gdb) n
9          i+=20;
(gdb) whatis j
type = int
(gdb) █
```

Figure 29: The *whatis* command

We can see that the **whatis** command gives the type of the variable **j**. Alternately, we can also use the **what** command.

```
(gdb) what j
type = int
(gdb) █
```

Figure 30: The *what* command

4 Advanced commands

4.1 Reverse debugging

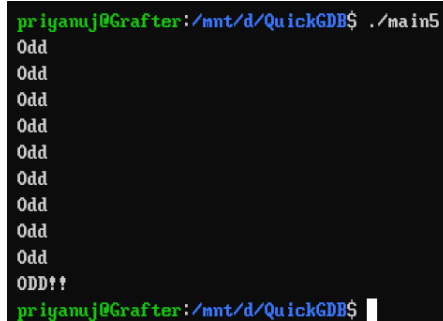
Reverse debugging is a feature in the GNU Debugger(GDB) that allows developers to step backward through the execution of a program, effectively rewinding the debugging session.

This capability is valuable for understanding the cause of a bug or error by allowing us to trace the execution history leading up to the issue.

4.2 A program

```
1 #include "black_box.h"
2 #include "fun.h"
3
4 int main(void) {
5     int x=16;
6
7     mystery(x, 1);
8     if(x%2)
9         fun(x);
10    else
11        funner(x);
12
13    return 0;
14 }
```

The output of this program is:



```
priyanuj@Grafter:/mnt/d/QuickGDB$ ./main5
Odd
Odd
Odd
Odd
Odd
Odd
Odd
Odd
Odd
Odd
Odd
Odd
Odd
ODD!?
```

Figure 31: Output of `main5.cpp`

4.2.1 The *target record-full* command

This command is used to enable the **record and replay** feature. This feature allows GDB to record the entire execution of a program, including all inputs and non-deterministic events, and then replay it later for analysis or debugging. Section **A program** shows us a program which performs some operations on a variable **x**.

Here are the definitions of the functions **mystery()**, **fun()** and **funner()**:

mystery()

```
1 #include<iostream>
2
3 void mystery(int& x, int incr) {
4     x+=incr;
5 }
```

fun() and funner()

```
1 #include<iostream>
2
3 using namespace std;
4
5 void fun(int x) {
6     for(int i=0; i<(x/2)+1; i++)
7         cout<<"Odd\n";
8     cout<<"ODD!!\n";
9 }
10
11 void funner(int x) {
12     for(int i=0; i<(x/2); i++)
13         cout<<"Even\n";
14     cout<<"EVEN!!\n";
15 }
```

We begin with the command:

```
(gdb) target record-full
```

After executing this command, it seems like nothing has happened:

```
(gdb) break main
Breakpoint 1 at 0x11b5: file main5.cpp, line 4.
(gdb) r
Starting program: /mnt/d/QuickGDB/main5
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at main5.cpp:4
4      int main(void) {
(gdb) target record-full
(gdb)
```

Figure 32: The *target record-full* command

Now, we can use commands like **reverse-next** to go back a step in the program execution like so:

```
Breakpoint 1, main () at main5.cpp:4
4      int main(void) {
(gdb) target record-full
(gdb) n
5          int x=16;
(gdb) n
7          mystery(x, 1);
(gdb)
8          if(x%2)
(gdb) reverse next
Ambiguous command "reverse next": reverse-continue, reverse-finish, reverse-next, reverse-nexti, reverse-search, reverse-step...
(gdb) reverse-next
7          mystery(x, 1);
(gdb)
```

Figure 33: The *reverse-next* command

As seen from figure, gdb completed execution till **mystery(x, 1);** and stopped at line **if(x%2)**. When we used the **reverse-next** command, the execution went one step backwards to line **mystery(x, 1);**. With the reversal of execution, the values of variables also get restored if they were modified.

```
7          mystery(x, 1);
(gdb) p x
$3 = 16
```

Figure 34: Value of **x** restored

We can see that when the execution reversed back by one step to **mystery(x, 1);**, the value of **x** also got restored.

To prove that it really worked:

```
(gdb) n
7          mystery(x, 1);
(gdb) p x
$5 = 16
(gdb) n

No more reverse-execution history.
main () at main5.cpp:8
8          if(x%2)
(gdb) p x
$6 = 17
(gdb) rn
7          mystery(x, 1);
(gdb) p x
$7 = 16
(gdb) █
```

Figure 35: Notice the values of `x`

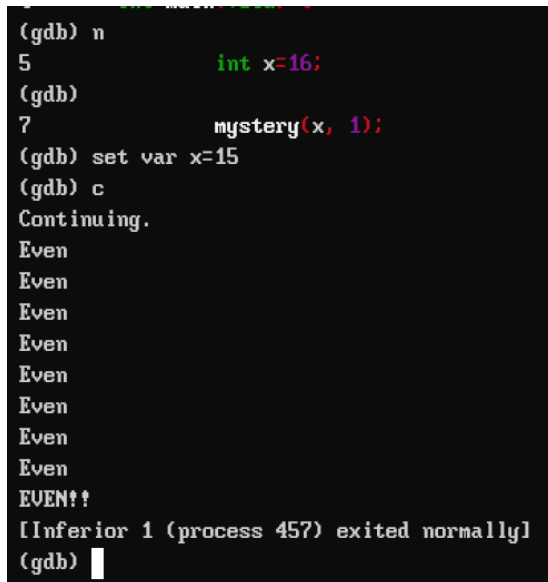
NOTE: `print` command is abbreviated as `p`. `reverse-next` is abbreviated as `rn`.

4.2.2 The `set var` command

This command is used to change the value of a variable during a debugging session i.e. while GDB is debugging a program.

Now, we will set the value of `x` to a different value.

```
(gdb) set var x=15
```



```
(gdb) n
5          int x=16;
(gdb)
7          mystery(x, 1);
(gdb) set var x=15
(gdb) c
Continuing.
Even
Even
Even
Even
Even
Even
Even
Even
Even
EVEN!!
[Inferior 1 (process 457) exited normally]
(gdb)
```

Figure 36: Changing the value of `x` to see how the output changes

From the figure above, we can see that the execution stopped at the line `mystery(x, 1);`. This function basically increments `x` by 1. So before this line got executed we set the value of `x` to 15. Then when we used the **continue** command and hence we get the output.

5 Sources

5.1 Youtube Videos

- GDB tutorial: <https://www.youtube.com/watch?v=svG60PyKsrw>