How to Use GDB to Debug Programs in Ubuntu 20.04

GNU Debugger (GDB) is an open-source debugger for GNU Systems. The debugger is portable and can be used for multiple languages as C/C++ and Fortran. It can be used for debugging programs by observing their states on specific breakpoints and even altering the flow or values for further execution. Usually, the GDB runs in command-line but several GUI has been developed for it as well.

In this article, we explore how to debug C programs using GDB in Ubuntu 20.04 LTS (Focal Fossa).

Prerequisites

- Ubuntu 20.04 system
- User with sudo privileges for renaming multiple files.

Note: The commands discussed in this article have been tested on **Ubuntu 20.04 LTS (Focal Fossa).**

Installing Packages

Install prerequisite GN packages for compiling and debugging. Run the following command in the terminal:

```
$ sudo apt install gcc gdb -y
```

C-Program example for debugging

Code

Before running, a program needs to be compiled. We are going to compile the following C code in the file *main.c*.

```
#include <stdio.h>

int main() {

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

Compile C Program using GCC

Usually, a C code is compiled in GCC using the following command:

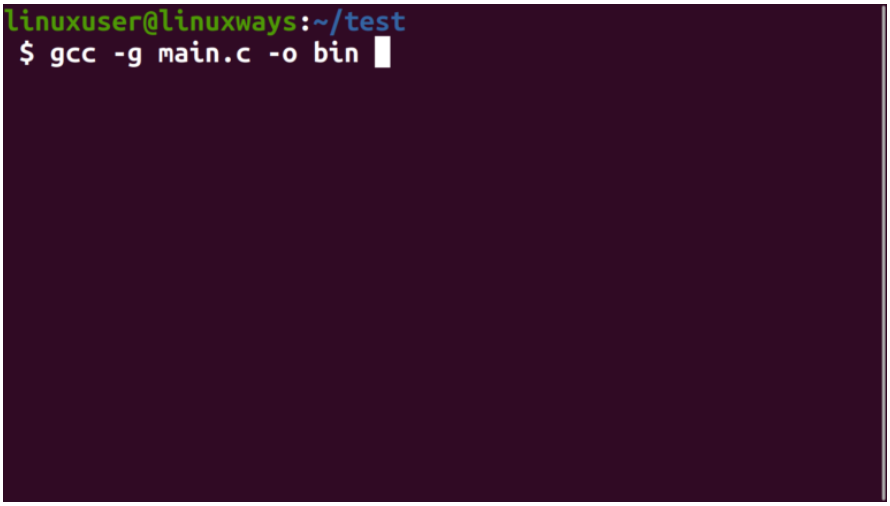
```
$ gcc main.c -o bin
```

Another argument needs to be provided to include *symbols* in the binary. These *symbols* are used by GDB to track and debug the program. Run the following command in terminal to compile the C code:

```
$ gcc -g main.c -o bin
```


About This Site

Vitux.com aims to become a Linux compendium with lots of unique and up to date tutorials.

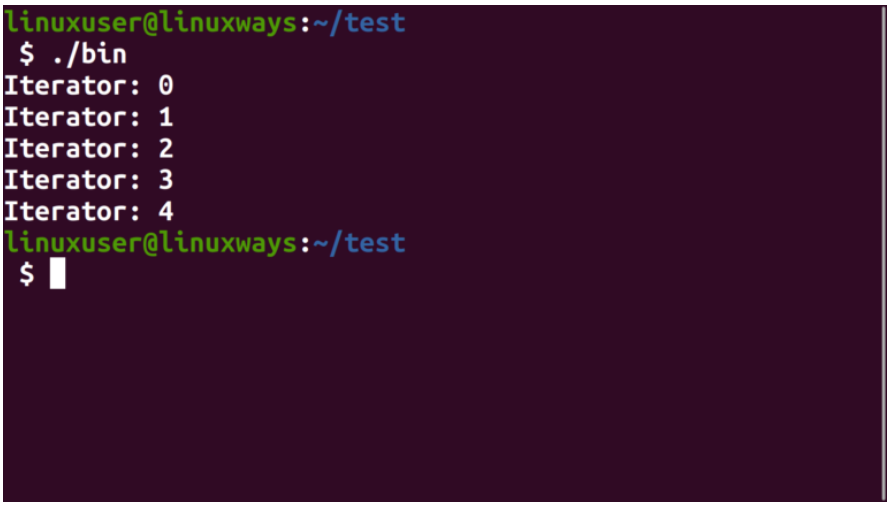


An executable file named *bin* will appear.

Execute the test program

The binary file named *bin* can be executed like any other executable file on a command-line interface. Use the following command to run it in terminal:

```
$ ./bin
```



Latest Tutorials

Directory structure of the Linux operating system explained

How to Install Webmin on CentOS 8 and Rocky Linux 8

How to format a harddisk partition with BTRFS on Ubuntu 20.04

How to speed-up an Ansible Playbook

How to install Reveal.js on Ubuntu 20.04

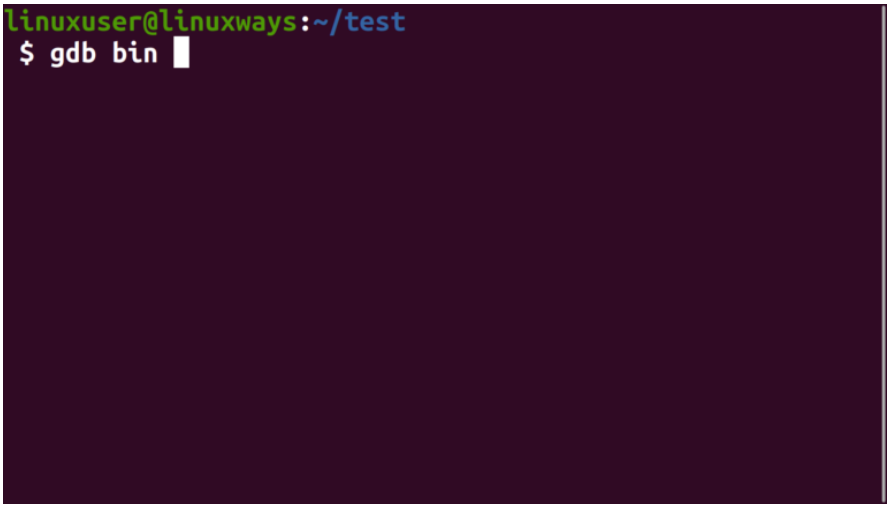
The output of the code will appear.

Debugging an Application on Linux

Initiate Debugger

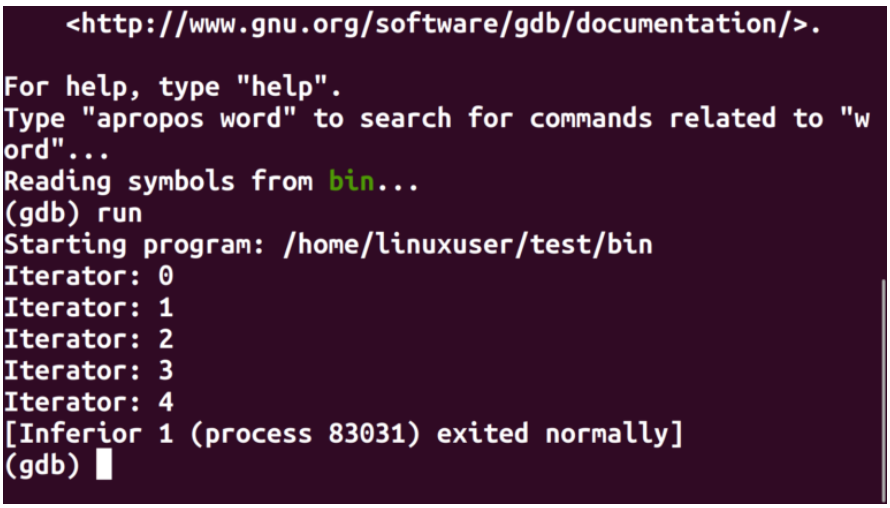
Run the GDB utility using following command in the terminal:

```
$ gdb bin
```



Press *enter.* The console for GDB terminal will appear. Enter the *run* command in this console to run the executable provided to the utility as an argument.

```
(gdb) run
```

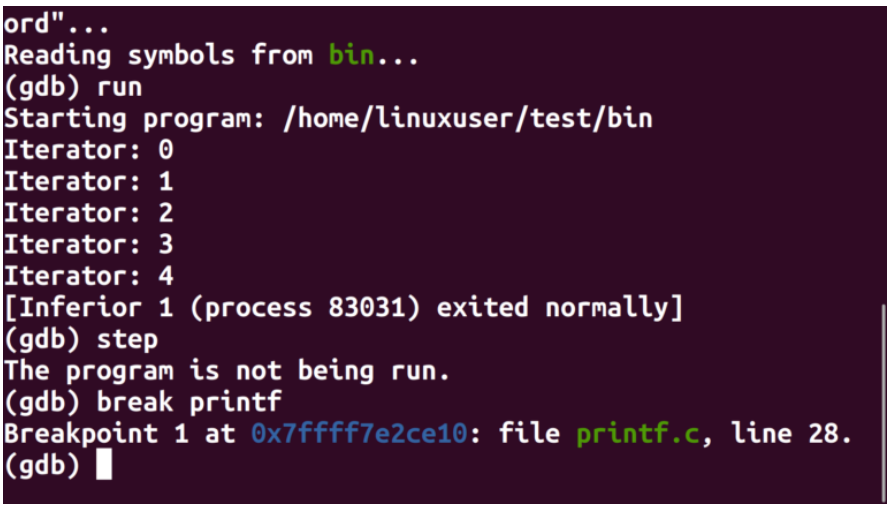


Debug Breakpoints

Add Breakpoints

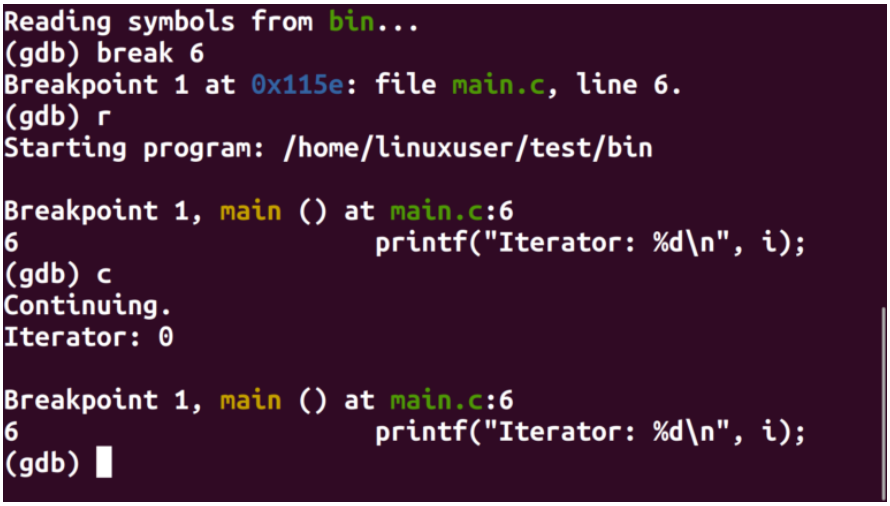
Breakpoints can be added in several ways. We will be adding a breakpoint on the *printf* function in our code. Run the following command in terminal to add a breakpoint:

```
(gdb) break printf
```



Alternatively, a line number can be used to add a breakpoint as well.

```
(gdb) break 6
```



Enter the *run* command and the program will stop at the breakpoint.

```
The program is not being run.
(gdb) break printf
Breakpoint 1 at 0x7ffff7e2ce10: file printf.c, line 28.
(gdb) run
Starting program: /home/linuxuser/test/bin

Breakpoint 1, __printf (
    format=0x555555556004 "Iterator: %d\n")
    at printf.c:28
28      printf.c: No such file or directory.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) n
Program not restarted.
(gdb)
```

Step through Breakpoints

Use the command *continue* to continue the execution of the program.

```
(gdb) continue
```

```
Iterator: 0

Breakpoint 1, __printf (
    format=0x555555556004 "Iterator: %d\n")
    at printf.c:28
28      printf.c: No such file or directory.
(gdb) continue
Continuing.
Iterator: 1

Breakpoint 1, __printf (
    format=0x555555556004 "Iterator: %d\n")
    at printf.c:28
28      in printf.c
(gdb)
```

There are two other commands for different purposes of continuing the execution of the program:

- Step: steps through the next machine instruction.
- Next: steps to through the next line of code.

Abbreviations of commands can also be used. Like abbreviation of *continue* command is c.

```
(gdb) c
```

```
    at printf.c:28
28      in printf.c
(gdb) c
Continuing.
Iterator: 3

Breakpoint 1, __printf (
    format=0x555555556004 "Iterator: %d\n")
    at printf.c:28
28      in printf.c
(gdb) c
Continuing.
Iterator: 4
[Inferior 1 (process 83710) exited normally]
(gdb)
```

Information About Breakpoints

Information about breakpoints can be observed using *info* command of *gdb*. Run the following command the terminal:

```
(gdb) info breakpoints
```

```
Breakpoint 1 at 0x115e: file main.c, line 6.
(gdb) r
Starting program: /home/linuxuser/test/bin

Breakpoint 1, main () at main.c:6
6                       printf("Iterator: %d\n", i);
(gdb) watch i
Hardware watchpoint 2: i
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000055555555515e in ma
in at main.c:6
        breakpoint already hit 1 time
2       hw watchpoint  keep y                      i
(gdb)
```

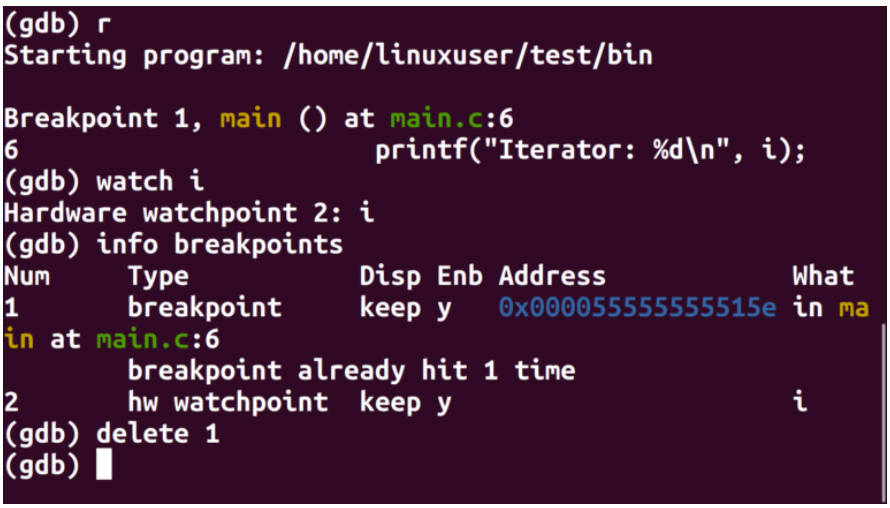
The information about breakpoints will appear.

Note: The number on the left of the breakpoint is used to refer to it by other commands.

Delete Breakpoints

A breakpoint can be deleted using the *delete* command and by referring to the breakpoint number observed in the output of the *info* utility.

```
(gdb) delete 1
```



Now the breakpoint has been deleted and if run, the program will execute straight to the end.

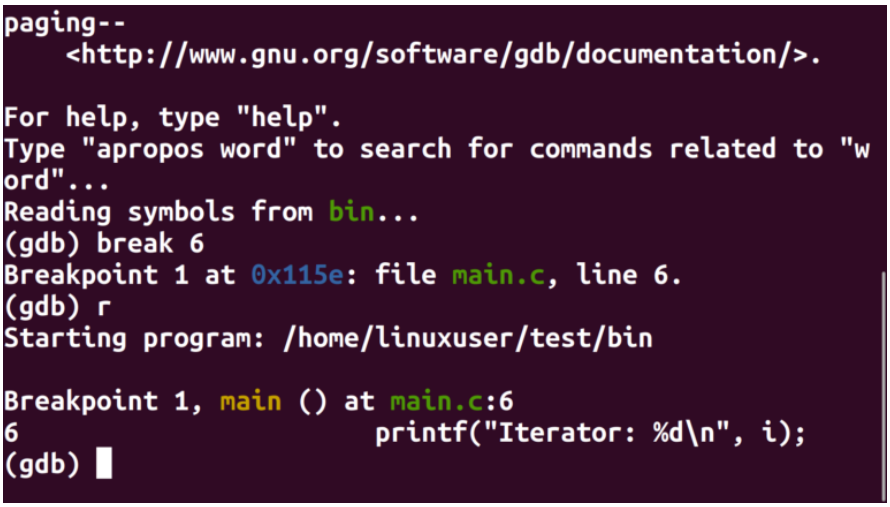
Watch Variables

Variables can be watched using the watch utility. First, we need to enter the scope in which the variable exists. For this purpose, add a breakpoint first using the following command:

```
(gdb) break 6
```

Then run the code that hits this breakpoint.

```
(gdb) r
```



Now we are in the loop where the variable *i* exists.

The *watch* command will be used to observe the previous and new value of the variable *i* in the loop.

```
(gdb) watch i
```

```
For help, type "help".
Type "apropos word" to search for commands related to "w
ord"...
Reading symbols from bin...
(gdb) break 6
Breakpoint 1 at 0x115e: file main.c, line 6.
(gdb) r
Starting program: /home/linuxuser/test/bin

Breakpoint 1, main () at main.c:6
6                       printf("Iterator: %d\n", i);
(gdb) watch i
Hardware watchpoint 2: i
(gdb)
```

Now the breakpoint generated by *watch* command will appear in the list of breakpoints as well. The list of breakpoints can be shown using the following command:

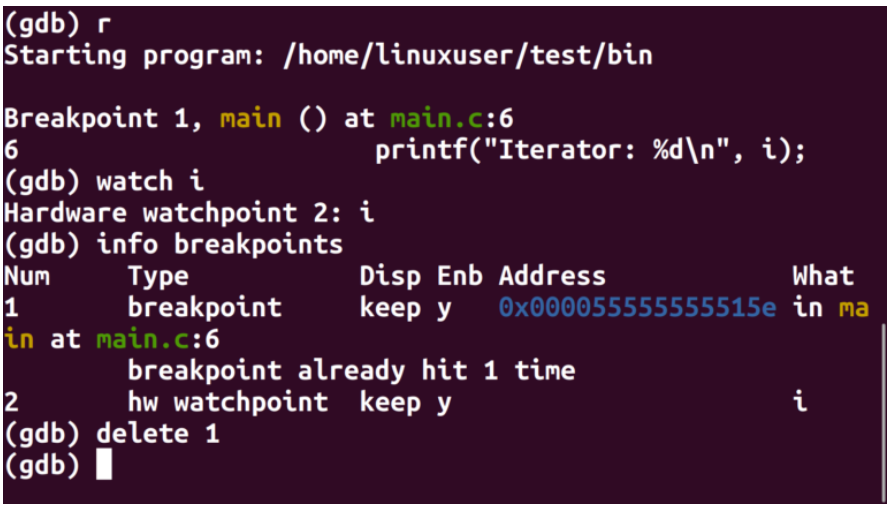
```
(gdb) info breakpoints
```

```
Breakpoint 1 at 0x115e: file main.c, line 6.
(gdb) r
Starting program: /home/linuxuser/test/bin

Breakpoint 1, main () at main.c:6
6                       printf("Iterator: %d\n", i);
(gdb) watch i
Hardware watchpoint 2: i
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000055555555515e in ma
in at main.c:6
        breakpoint already hit 1 time
2       hw watchpoint  keep y                      i
(gdb)
```

Moreover, we do not need the breakpoint inserted earlier. It can be easily removed using the following command:

```
(gdb) delete 1
```



Now if continued, the code will view values whenever the variable has changed the value and show both old and new values.

```
(gdb) c
```

Further iterations of the program can be observed as well, using the same command.

```
Old value = 1
New value = 2
0x0000555555555178 in main () at main.c:5
5               for (int i=0; i<5; ++i) {
(gdb) c
Continuing.
Iterator: 2

Hardware watchpoint 2: i

Old value = 2
New value = 3
0x0000555555555178 in main () at main.c:5
5               for (int i=0; i<5; ++i) {
(gdb)
```

Quit Debugger

Run the following command in the terminal to exit the debugger.

```
(gdb) quit
```

```
(gdb) c
Continuing.
Iterator: 3

Breakpoint 1, __printf (
    format=0x555555556004 "Iterator: %d\n")
    at printf.c:28
28      in printf.c
(gdb) c
Continuing.
Iterator: 4
[Inferior 1 (process 83710) exited normally]
(gdb) quit
linuxuser@linuxways:~/test
 $
```

This close *gdb* utility and the default command-line prompt will appear.

Conclusion

In this article, we explored how to run and break a program in GDB. Moreover, it was also configured to break itself when the value of a variable has changed. We hope you can easily debug your programs in GDB after following this article.

Karim Buzdar February 19, 2021 Linux, Shell, Ubuntu

← How to Install Gradle Build-Tool on Ubuntu 20.04

Killing frozen applications in Ubuntu 20.04 →