

# GDB (GNU Debugger)

## Contents

1. Introduction to GDB .....	3
Compiling for Debugging.....	3
Starting GDB .....	4
Basic Commands.....	4
2. Breakpoints.....	5
List source code: .....	6
3. Program Execution Flow .....	7
4. Passing Command-Line Arguments to Executables.....	8
Using --args when launching GDB.....	8
Using run command with arguments after entering GDB .....	8
Using set args before running.....	9
Summary Table .....	9
5. Examining Execution.....	10
Print Values .....	10
Examine Memory .....	11
Type Inspection.....	11
Stack Traces & Debugging Crashes .....	12
6. Call Stack & Frames .....	13
INFO Commands Summary .....	13
7. Conditional Breakpoints .....	14
Watchpoints .....	14

8.	TUI (Text User Interface) .....	15
9.	Logging GDB Output.....	15
10.	Attaching to a Running Process .....	15
11.	GDB start Command .....	16
	Auto-Run Commands on Breakpoint.....	16
12.	Summary of Common GDB Features.....	17

## 1. Introduction to GDB

- **GDB** stands for *GNU Debugger*.
- It helps in debugging **binary executables** created during compilation.
- Main Features:
  - Step-by-step execution.
  - Setting breakpoints (function/line).
  - Conditional execution control.
  - Examining variables and memory.
  - Viewing the call stack.

### Compiling for Debugging

- Use the **-g** flag to include **debug symbols**:

```
total 8
-rw-rw-r-- 1 nishirh nishirh 225 Jul  2 03:11 source.cpp
drwxrwxr-x 2 nishirh nishirh 4096 Jul  2 03:11 output
• nishirh@ubuntu:~/Programming$ ls -ltr output/
total 140
-rwxrwxr-x 1 nishirh nishirh 140592 Jul  2 03:11 source
• nishirh@ubuntu:~/Programming$ g++ -g -o output/source source.cpp
• nishirh@ubuntu:~/Programming$ ls -ltr output/
total 68
-rwxrwxr-x 1 nishirh nishirh 69216 Jul  2 03:12 source
```

Size of the binary before including debug symbols.

Use the **-g** flag to include debug symbols.

Size of the binary after including debug symbols.

- This embeds:
  - Symbol names (variables/functions)
  - Types
  - File names
  - Line numbers
- Increases executable size due to metadata.

## Starting GDB

- Two ways to start gdb for debugging.

### 1. Direct:

```
nishirh@ubuntu:~/Programmings$ gdb ./output/a.out
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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:
<http://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 ./output/a.out...
(gdb) █
```

`gdb ./a.out`

### 2. Load inside gdb:

```
nishirh@ubuntu:~/Programmings$ gdb
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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:
<http://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".
(gdb) file output/a.out
Reading symbols from output/a.out...
(gdb) █
```

Note: To start GDB in quite mode, use `-q` option.

## Basic Commands

- GDB is a command line interface.

Command	Shortcut	Description
run	r	Start program execution
quit	q	Exit GDB
help [command]		Get help for commands

## 2. Breakpoints

- The main purpose of GDB is to stop, observe and proceed.
- Breakpoints can be used to stop the program in the middle at a designated point.
- The simplest way of keeping a breakpoint is either using function name or a line number.
- Syntax:
  - By **function**: **break main**
  - By **line** number: **break 12**
  - Short: **b main** or **b 12**
- To find out what functions are available, use

info functions

```
nishirh@ubuntu:~/Programming$ gdb -q output/a.out
Reading symbols from output/a.out...
(gdb) info functions
All defined functions:

File source.cpp:
5:   unsigned int factorial(unsigned int);
13:   int main();
static void __GLOBAL_sub_I_29factorialj();
static void __static_initialization_and_destruction_0(int, int);

Non-debugging symbols:
0x0000000000001000  _init
0x0000000000001000  _cxa_finalize@plt
0x0000000000001000  _cxa_atexit@plt
0x00000000000010a0  std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char const*)@plt
0x00000000000010b0  std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::basic_ostream<char, std::char_traits<char> >&, char)@plt
0x00000000000010c0  std::ostream::operator<<(unsigned int)@plt
0x00000000000010d0  std::ios_base::Init::Init()@plt
0x00000000000010e0  _start
0x0000000000001110  deregister_tm_clones
0x0000000000001140  register_tm_clones
0x0000000000001180  _do_global_dtors_aux
0x00000000000011c0  frame_dummy
0x00000000000012f0  _libc_csu_init
0x0000000000001360  _libc_csu_fini
0x0000000000001368  _fini
(gdb) █
```

- To list all the breakpoints, use

info breakpoints

```
(gdb) b main
Breakpoint 1 at 0x11fe: file source.cpp, line 13.
(gdb) info breakpoints
Num      Type             Disp Enb Address            What
1        breakpoint       keep y   0x00000000000011fe in main() at source.cpp:13
(gdb) █
```

- To delete a breakpoint, use

delete 1 or d 1

```
(gdb) info breakpoints
Num    Type             Disp Enb Address                  What
1      breakpoint      keep y   0x00000000000001fc in main() at source.cpp:13
2      breakpoint      keep y   0x00000000000001c9 in factorial(unsigned int) at source.cpp:5
(gdb) d 1
(gdb) info breakpoints
Num    Type             Disp Enb Address                  What
2      breakpoint      keep y   0x00000000000001c9 in factorial(unsigned int) at source.cpp:5
(gdb) █
```

List source code:

- To list the source code while debugging, use

```
list          # next 10 lines
list 1,10     # specific range
list main     # around function
```

```
(gdb) list 1
1      #include<iostream>
2
3      using namespace std;
4
5      unsigned int factorial(unsigned int n) {
6          if (n == 0 || n == 1) {
7              return 1;
8          }
9          return n * factorial(n - 1);
10     }
(gdb) list 5, 10
5      unsigned int factorial(unsigned int n) {
6          if (n == 0 || n == 1) {
7              return 1;
8          }
9          return n * factorial(n - 1);
10     }
(gdb) list main
8      }
9      return n * factorial(n - 1);
10     }
11
12
13     int main() {
14         unsigned int loop = 50;
15         while(loop--){
16             cout << "Factorial of " << loop << " is: "
17             << factorial(loop) << '\n';
(gdb) █
```

### 3. Program Execution Flow

- Whenever a breakpoint is hit during execution, the program stops.
- At that point, you can examine variables, memory, the call stack, and step through your code using GDB commands.

Command	Description
run	Starts program (from beginning)
continue / c	Resumes until next breakpoint
next / n	Executes next line, skips over function calls
step / s	Steps into function
finish	Runs until current function returns
[Enter]	Repeats last command

```
nishirh@ubuntu:~/Programming$ gdb -q ./output/print.out
Reading symbols from ./output/print.out...
(gdb) list
1  #include <iostream>
2  using namespace std;
3
4  void print(const char* str) {
5      cout << str << '\n';
6  }
7
8  int main() {
9      const char* message = "Hello, World!";
10     print(message);
(gdb) b 5
Breakpoint 1 at 0x11b9: file print.cpp, line 5.
(gdb) b 10
Breakpoint 2 at 0x11f3: file print.cpp, line 10.
(gdb) info b
Num    Type             Disp Enb Address            What
1      breakpoint       keep y   0x00000000000011b9 in print(char const*) at print.cpp:5
2      breakpoint       keep y   0x00000000000011f3 in main() at print.cpp:10
(gdb) r
Starting program: /home/nishirh/Programming/output/print.out
Breakpoint 2, main() at print.cpp:10
10     print(message);
(gdb) s
print (str=0x55555555270 <__libc_csu_init> "\363\017\036\372AWL\215=(003+ ) at print.cpp:4
4     void print(const char* str) {
(gdb) n
Breakpoint 1, print (str=0x555555556005 "Hello, World!") at print.cpp:5
5     cout << str << '\n';
(gdb) c
Continuing.
Hello, World!
[Inferior 1 (process 79781) exited normally]
(gdb)
```

Run GDB in quite mode.

List the source code in GDB

Add a breakpoint at line 5

Add a breakpoint at line 10

List all the breakpoints.

Run the program and stop at the breakpoint.

Steps into function 'print'

Executes next line, skips over function calls

Resumes until next breakpoint

## 4. Passing Command-Line Arguments to Executables

- When debugging a C or C++ program with GDB, if your **executable expects command-line arguments**, there are **multiple ways to provide them**.

### Using `--args` when launching GDB

- This method allows you to pass the arguments while starting GDB.  
`gdb --args ./my_program source_code 6`
- The `--args` tells GDB to treat everything after it as the executable and its arguments.

```
nishirh@ubuntu:~/Programming$ gdb -q --args output/a.out 6
Reading symbols from output/a.out...
(gdb) r
Starting program: /home/nishirh/Programming/output/a.out 6
Factorial of 5 is 120
Factorial of 4 is 24
Factorial of 3 is 6
Factorial of 2 is 2
Factorial of 1 is 1
Factorial of 0 is 1
[Inferior 1 (process 4217) exited normally]
(gdb)
```

### Using `run` command with arguments after entering GDB

- If you've already started GDB and loaded your executable, then you can pass arguments using the `run` command.

```
nishirh@ubuntu:~/Programming$ gdb -q ./output/a.out
Reading symbols from ./output/a.out...
(gdb) r 6
Starting program: /home/nishirh/Programming/output/a.out 6
Factorial of 5 is 120
Factorial of 4 is 24
Factorial of 3 is 6
Factorial of 2 is 2
Factorial of 1 is 1
Factorial of 0 is 1
[Inferior 1 (process 4876) exited normally]
(gdb)
```



## Using set args before running

- You can also set the arguments first, then run:

```
nishirh@ubuntu:~/Programming$ gdb -q ./output/a.out
Reading symbols from ./output/a.out...
(gdb) set args 6
(gdb) r
Starting program: /home/nishirh/Programming/output/a.out 6
Factorial of 5 is 120
Factorial of 4 is 24
Factorial of 3 is 6
Factorial of 2 is 2
Factorial of 1 is 1
Factorial of 0 is 1
[Inferior 1 (process 6668) exited normally]
(gdb)
```

## Summary Table

Method	Command	When to Use
Using --args	<code>gdb --args ./my_program arg1 arg2</code>	At GDB launch
Using run	<code>gdb ./my_program → run arg1 arg2</code>	After loading GDB
Using set args + run	<code>gdb ./my_program → set args arg1 arg2 → run</code>	Set once, reuse multiple run calls

## 5. Examining Execution

### Print Values

- `print var` or `p var`
- Show address: `p &var`
- Format options:
  - Decimal: `p /d var`
  - Hex: `p /x var`
  - Octal: `p /o var`
  - Binary (if supported): `p /t var`

```
nishirh@ubuntu:~/Programming$ g++ -g -o ./output/print.out print.cpp
nishirh@ubuntu:~/Programming$ gdb -q ./output/print.out
Reading symbols from ./output/print.out...
(gdb) list
1      #include <iostream>
2
3      int main()
4      {
5
6          int a = 42;
7          int b = 255;
8          int c = 7;
9
10         std::cout << "Test GDB print commands!" << std::endl;
(gdb) b 10
Breakpoint 1 at 0x11ca: file print.cpp, line 10.
(gdb) r
Starting program: /home/nishirh/Programming/output/print.out

Breakpoint 1, main () at print.cpp:10
10         std::cout << "Test GDB print commands!" << std::endl;
(gdb) p a
$1 = 42
(gdb) p b
$2 = 255
(gdb) p c
$3 = 7
(gdb) p &a
$4 = (int *) 0x7fffffffdfc4
(gdb) p &b
$5 = (int *) 0x7fffffffdfc8
(gdb) p &c
$6 = (int *) 0x7fffffffdfcc
(gdb) p /d b
$7 = 255
(gdb) p /x b
$8 = 0xff
(gdb) p /o b
$9 = 0377
(gdb) p /t b
$10 = 11111111
(gdb)
```

## Examine Memory

- Syntax: `x /<count><format><size> address`
  - e.g., `x/4xb &i` → 4 hex bytes at `i`
- Useful format specifiers:
  - `x` – Hex
  - `d` – Decimal
  - `s` – String
  - `b` – Byte
  - `w` – Word (4 bytes)

```
Breakpoint 1, main () at print.cpp:10
10      std::cout << "Test GDB print commands!" << std::endl;
(gdb) x/4xb &a
0x7fffffffdfc4: 0x2a    0x00    0x00    0x00
(gdb) x/4xb &b
0x7fffffffdfc8: 0xff    0x00    0x00    0x00
(gdb) x/4w &b
0x7fffffffdfc8: 0x000000ff 0x00000007 0x00000000 0x00000000
(gdb) x/1dw &b
0x7fffffffdfc8: 255
(gdb) █
```

## Type Inspection

- `ptype var` → Show type of variable
- `ptype &var` → Pointer type
- `ptype main` → Function signature

```
(gdb) ptype a
type = int
(gdb) ptype &a
type = int *
(gdb) ptype main
type = int (void)
(gdb) █
```

## Stack Traces & Debugging Crashes

### Scenario: Segmentation fault

- Use `backtrace (bt)` to view call stack
- Use `frame <n>` to switch stack frame
- Use `list` to view surrounding source
- Use `print var` to inspect suspected variables

```
seg_fault.cpp > main()
1  #include <iostream>
2  #include <cstdlib>
3  using namespace std;
4
5  int main()
6  {
7      constexpr long long size = 1LL << 35; // 32 GiB
8      char *buff = (char*)malloc(size);
9      std::cin.getline(buff, 1024);
10
11     std::cout << "You entered: " << buff << std::endl;
12     return 0;
13 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
nishirh@ubuntu:~/Programming$ gdb -q ./output/seg.out
Reading symbols from ./output/seg.out...
(gdb) b 11
Breakpoint 1 at 0x122e: file seg_fault.cpp, line 11.
(gdb) r
Starting program: /home/nishirh/Programming/output/seg.out
Hello

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7e98dda in std::istream::getline(char*, long, char) () from /lib/x86_64-linux-gnu/libstdc++.so.6
(gdb) bt
#0  0x00007ffff7e98dda in std::istream::getline(char*, long, char) () from /lib/x86_64-linux-gnu/libstdc++.so.6
#1  0x000055555555522e in main () at seg_fault.cpp:9
(gdb) frame 1
#1  0x000055555555522e in main () at seg_fault.cpp:9
9      std::cin.getline(buff, 1024);
(gdb) p buff
$1 = 0x0
(gdb)
```

## 6. Call Stack & Frames

- **Call Stack:** Stack of function calls.
- **Frame:** Each function call creates a *frame* in the stack.
- **Frame info:**
  - Return address
  - Local variables
  - Arguments

### Commands:

- **backtrace** (or **bt**): Shows current call stack.
- **frame n**: Switch to specific frame number.
- **info frame**: Details like instruction pointer, stack pointer, local variable addresses.
- **list**: Displays source code of current frame.
- **print var**: Inspects local variable of selected frame.

### INFO Commands Summary

Command	Description
<b>info functions</b>	List all functions in symbol table
<b>info variables</b>	List global & static variables
<b>info locals</b>	List local variables of current frame
<b>info args</b>	Function arguments in current frame
<b>info breakpoints</b>	Show all breakpoints and watchpoints
<b>info registers</b>	Show all CPU register values
<b>info register RAX</b>	Show specific register value

## 7. Conditional Breakpoints

- Stop execution only **when a condition is met**.

- **Two Ways:**

- **Add with condition directly:**

`break 15 if i == 5`

- **Apply condition after setting:**

`break 15`

`condition <breakpoint-number> i == 5`

- Useful for skipping unnecessary iterations (e.g., in loops).
- `info breakpoints`: Show all breakpoints with their conditions.

## Watchpoints

- **Used for variables**, not line numbers.
- Execution stops when:
  - A value is **read**, **written**, or **both** (depending on watch type).

Watch Type	Trigger Condition	Command
<code>watch var</code>	On write to variable	<code>watch x</code>
<code>rwatch</code>	On read from variable	<code>rwatch x</code>
<code>awatch</code>	On read/write to variable	<code>awatch x</code>

- Must be in scope to apply.
- Use `disable n` / `enable n` to toggle watchpoint.
- Show all: `info breakpoints`

## 8. TUI (Text User Interface)

- Show source code while stepping through.
- Start with:

```
gdb -tui ./a.out
```

- Inside GDB:
  - Toggle TUI: Ctrl + X followed by A
  - Refresh TUI: Ctrl + L

### Benefits:

- Live source code view
- Visual breakpoints
- Highlight current line

## 9. Logging GDB Output

- Save session output to a file.

```
set logging on           # Start logging to gdb.txt
set logging off          # Stop logging
set logging file mylog.txt # Custom file name
```

## 10. Attaching to a Running Process

- Attach GDB to a live process by PID.

### Steps:

1. Get PID:

```
shell ps -ef | grep myapp
```

2. Attach:

```
attach <pid>
```

3. Detach (resume normal execution):

```
detach
```

- Requires root permissions (`sudo gdb`)

## 11. GDB start Command

- Shortcut for:

```
break main
```

```
run
```

- Just use:

```
Start
```

### Auto-Run Commands on Breakpoint

- Automate actions like printing variables when breakpoint hits.

```
break 15  
commands 1  
    print i  
    continue  
end
```



## 12. Summary of Common GDB Features

Category	Commands / Concepts
<b>Breakpoints</b>	break, info breakpoints, delete
<b>Run control</b>	run, continue, next, step, finish, start
<b>Call Stack</b>	backtrace, frame n, info frame
<b>Variable Inspect</b>	print, info locals, info args, info variables
<b>Memory Inspect</b>	x, x/s, x/4xb, etc.
<b>Assembly View</b>	disassemble main
<b>Registers</b>	info registers, p \$rax
<b>Conditional BP</b>	break 12 if i == 5, condition
<b>Watchpoints</b>	watch, rwatch, awatch
<b>Logging</b>	set logging on, set logging file log.txt
<b>TUI Mode</b>	gdb -tui, Ctrl + X, A, Ctrl + L
<b>Attach/Detach</b>	attach <pid>, detach
<b>Startup Helper</b>	start (shortcut for break main + run)