Here's a **complete, in-depth explanation of GBD** (likely **Global Buffering or Global Block Distribution**, depending on context) in the scope of **Parallel and Distributed Computing**, including examples and applications. Since "GBD" can have multiple meanings, I'll explain it in the **common HPC / parallel computing sense: Global Block Distribution**.

GBD (Global Block Distribution) in Parallel & Distributed Computing

1. Definition of GBD

GBD (Global Block Distribution) is a **data distribution strategy** used in parallel computing, where **large datasets are divided into blocks** and distributed across multiple **processors or nodes**.

It allows **parallel processing** of large datasets while maintaining **load balance** and **data locality**.

Exam-ready one-line definition:

GBD is a data distribution technique in parallel computing where a dataset is divided into blocks and distributed across processors to enable parallel execution and efficient load balancing.

---

2. Why GBD is Needed

- Large datasets may **not fit into memory of a single processor**
- Sequential processing is **slow**
- Parallel computing requires **data locality** to reduce communication overhead
- GBD ensures:
    - Even workload among processors
    - Efficient memory usage
    - Minimal inter-processor communication

---

3. GBD in Parallel & Distributed Computing

GBD is widely used in:

- **Matrix computations** (e.g., matrix multiplication, LU decomposition)
- **Finite element methods**
- **Image processing**
- **Scientific simulations**

Key Idea: Each processor works on its **local block**, reducing dependency on other processors and enabling **parallel computation**.

---

4. GBD Strategies

4.1 Block Distribution

- Data divided into **contiguous blocks**
- Each processor gets a **block**
- Example: Array of 16 elements distributed to 4 processors

```
Array: [0,1,2,...,15]
Processor 0: [0,1,2,3]
Processor 1: [4,5,6,7]
Processor 2: [8,9,10,11]
Processor 3: [12,13,14,15]
```

4.2 Cyclic Distribution

- Data distributed in a **round-robin fashion**
- Useful for **load balancing**

```
Processor 0: 0,4,8,12
Processor 1: 1,5,9,13
Processor 2: 2,6,10,14
Processor 3: 3,7,11,15
```

4.3 Block-Cyclic Distribution

- Combines **block and cyclic**
- Blocks of multiple elements distributed cyclically
- Example: Block of 2 elements per processor in cyclic fashion

---

5. GBD Example: Matrix Multiplication

Given: Two matrices `A` and `B`, want to compute `C = A*B` using 4 processors.

1. **Step 1: Divide matrix A into 4 blocks of rows**

```
P0 → rows 0-1
P1 → rows 2-3
P2 → rows 4-5
P3 → rows 6-7
```

2. **Step 2: Each processor computes partial matrix multiplication**

```
C_local = A_block * B
```

3. **Step 3: Combine results to form final matrix**

- Each processor sends computed `C_local` to master or shared memory
- Final `C` is assembled

Benefit: Computation is done **in parallel**, reducing total runtime.

---

6. Advantages of GBD

- **Parallelism:** Processors work on different blocks simultaneously
- **Load Balancing:** Blocks can be distributed evenly
- **Reduced Communication:** Data mostly local to processor
- **Scalability:** Works with many processors

---

7. Limitations of GBD

- **Data dependencies:** Some computations may require communication between blocks
- **Uneven workloads:** If data is irregular, some processors may be idle
- **Complex implementation** for dynamic datasets

---

8. GBD vs Other Parallel Distribution Methods

Feature	GBD	Row-wise/Column-wise	Cyclic	Block-Cyclic
Data Division	Blocks	Rows / Columns	Round-robin	Blocks in cyclic pattern
Load Balancing	Good for uniform data	Depends on matrix	Better for non-uniform data	Best for mixed workloads

Feature	GBD	Row-wise/Column-wise	Cyclic	Block-Cyclic
Communication	Low	Medium	Medium	Medium
Example	Parallel matrix multiplication	Row-based multiplication	Sparse matrices	Hybrid tasks

---

9. Real-World Use Cases

- **HPC Applications:** CFD (Computational Fluid Dynamics)
- **Scientific Computing:** Weather simulations, molecular dynamics
- **Parallel Linear Algebra Libraries:** ScaLAPACK, MPI-based matrix routines
- **Distributed Image/Video Processing**

---

10. Exam-Oriented Summary

Keywords:

- Global Block Distribution, parallel processing, load balancing, block/cyclic/block-cyclic, distributed memory

One-Paragraph Answer:

GBD (Global Block Distribution) is a data distribution strategy in parallel computing where a dataset is divided into blocks and distributed across processors or nodes. Each processor works on its local block, enabling parallel execution and reducing communication overhead. GBD is widely used in matrix computations, scientific simulations, and distributed data processing, and can be implemented in block, cyclic, or block-cyclic patterns depending on the application.

If you want, I can next create a **diagram showing GBD with block, cyclic, and block-cyclic distributions across multiple processors**, which makes **parallel computation and load balancing very easy to visualize**.

Do you want me to create that diagram?

The **GNU Debugger (GDB)** is primarily a command-line debugger for *serial* programs, but it offers limited, low-cost support for **multithreaded parallel** applications and **multiprocess distributed** programs.

GDB for Parallel Computing

GDB supports debugging **multithreaded applications** natively within a single GDB session.

- **Multithreading Models**: It works with shared memory models like OpenMP, where multiple threads share the same address space on a single machine.
- **Debugging Multithreaded Code**: You can set breakpoints and step through code in multithreaded programs. When a breakpoint is hit, you can inspect variables and control the execution of individual threads using GDB commands like `thread <ID>`, `break <filename>:<linenumber>`, `step`, and `next`.
- **Limitations**: GDB is generally not well-suited for debugging subtle timing events in large, complex parallel programs due to the inherent challenges of observing concurrent execution in real-time without altering behavior.

GDB for Distributed Computing

GDB's support for distributed computing (multiple independent processes, potentially across a network, using message passing like MPI) is more limited.

- **Multiple Instances**: Standard GDB can only debug one process at a time. To debug a distributed application, you typically need to use a separate GDB instance for each process.
- **GDBserver and Remote Debugging**: You can use `gdbserver` on remote hosts to start or attach to processes, and then connect to them from a local GDB session.
- **Specialized Extensions/Distributions**: Debugging large-scale distributed systems effectively often requires specialized tools or extensions.
  - **Intel® Distribution for GDB***: This distribution provides a unified debugging experience for cross-platform parallel and threaded applications (C, C++, SYCL, OpenMP, Fortran), including debugging across CPU and GPU code in a single session. It offers features to handle thousands of threads simultaneously.
  - **Custom Debuggers**: Researchers and developers have built custom debuggers or extensions on top of GDB to better support specific parallel libraries (like the DASH template library using MPI3) and manage the complexities of distributed states and communication patterns.

In summary, while the base GDB tool can be used for fundamental debugging tasks in these environments, specialized tools and techniques are often necessary for efficient and robust debugging of complex, large-scale parallel and distributed applications.

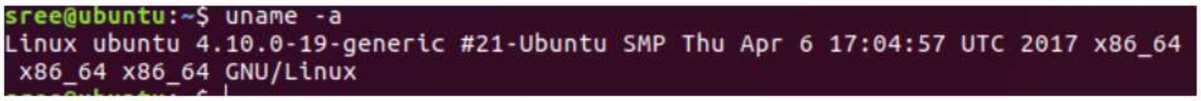
GDB (Step by Step Introduction)

Last Updated : 10 Jan, 2025

GDB stands for GNU Project Debugger and is a powerful debugging tool for C (along with other languages like C++). It helps you to poke around inside your C programs while they are executing and also allows you to see what exactly happens when your program crashes. GDB operates on executable files which are binary files produced by the compilation process.

For demo purposes, the example below is executed on a Linux machine with the below specs.

```
uname -a
```



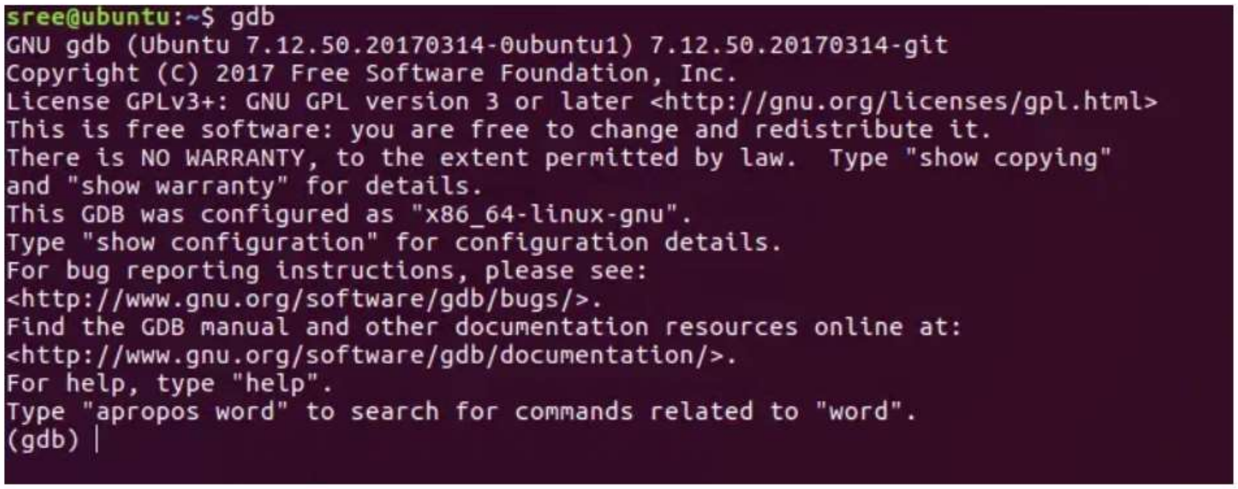
```
sree@ubuntu:~$ uname -a
Linux ubuntu 4.10.0-19-generic #21-Ubuntu SMP Thu Apr 6 17:04:57 UTC 2017 x86_64
 x86_64 x86_64 GNU/Linux
```

Let's learn by doing: -

Start GDB

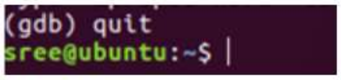
Go to your Linux command prompt and type "gdb".

```
gdb
```



```
sree@ubuntu:~$ gdb
GNU gdb (Ubuntu 7.12.50.20170314-0ubuntu1) 7.12.50.20170314-git
Copyright (C) 2017 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) |
```

Gdb open prompt lets you know that it is ready for commands. To exit out of gdb, type quit or q.

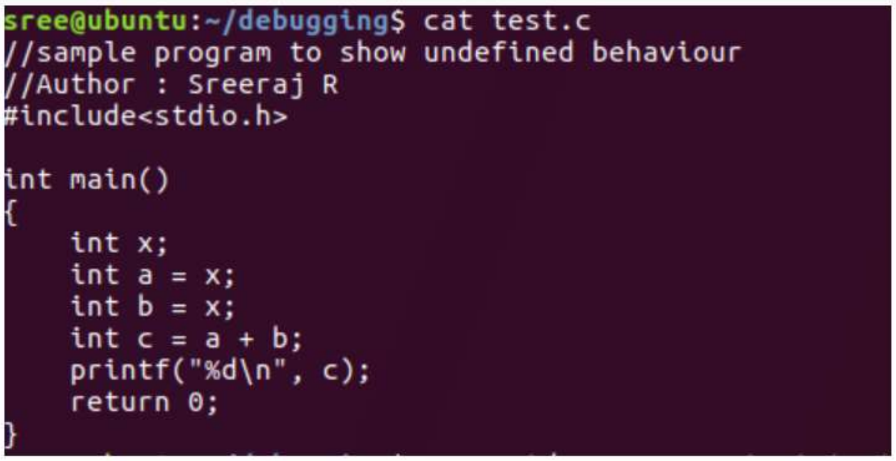


```
(gdb) quit
sree@ubuntu:~$ |
```

To quit

Compile the code

Below is a program that shows undefined behavior when compiled using C99. [caption id="attachment_646191" align="alignnone"]



```
sree@ubuntu:~/debugging$ cat test.c
//sample program to show undefined behaviour
//Author : Sreeraj R
#include<stdio.h>

int main()
{
    int x;
    int a = x;
    int b = x;
    int c = a + b;
    printf("%d\n", c);
    return 0;
}
```

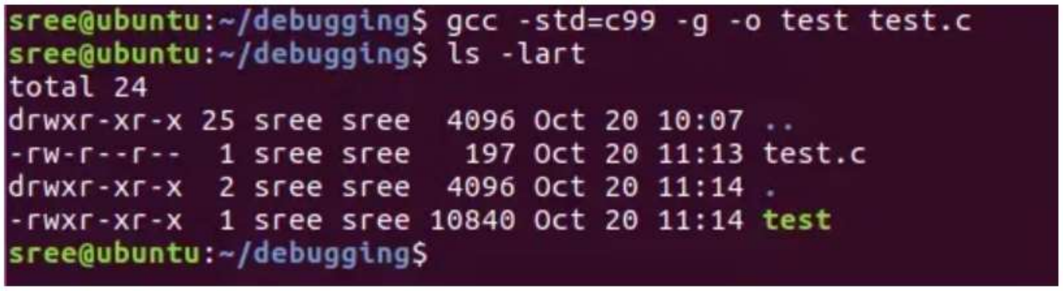
cat test.c

Note: If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate, where the indeterminate value is either an unspecified value or a trap representation.

Now compile the code. (here test.c). **g flag** means you can see the proper names of variables and functions in your stack frames, get line numbers and see the source as you step around in the executable. **-std=C99 flag** implies use standard C99 to compile the code. **-o flag** writes the build output to an output file.

output file.

```
gcc -std=c99 -g -o test test.C
```



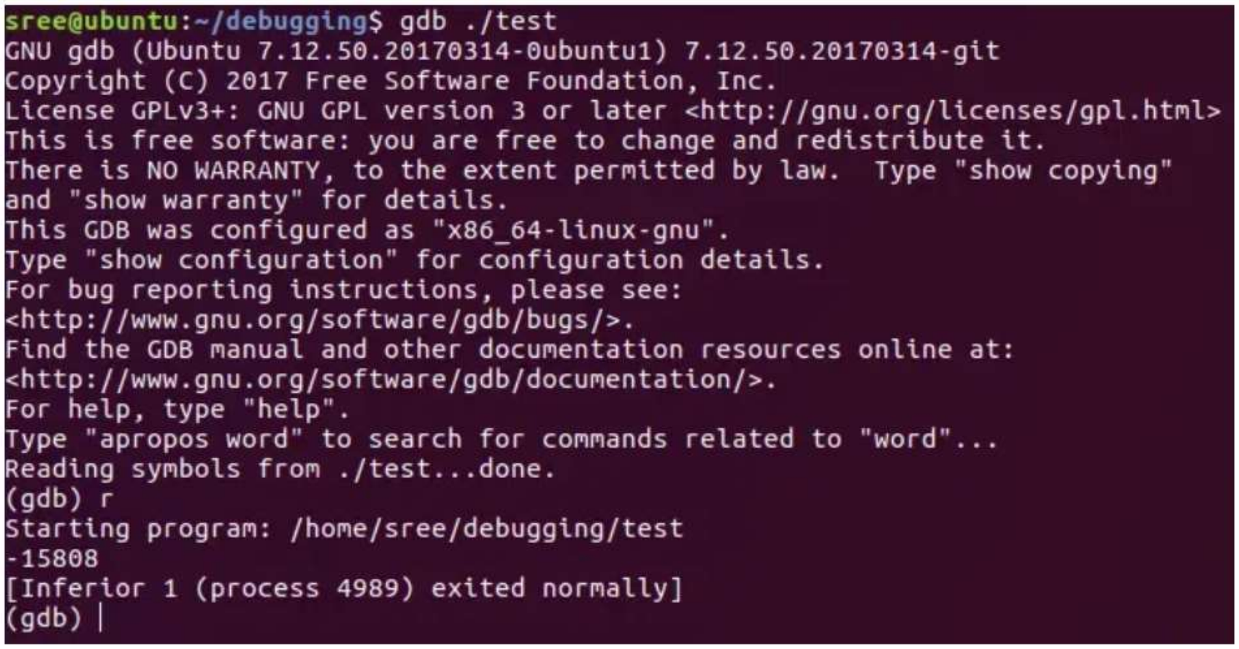
```
sree@ubuntu:~/debugging$ gcc -std=c99 -g -o test test.c
sree@ubuntu:~/debugging$ ls -lart
total 24
drwxr-xr-x 25 sree sree  4096 Oct 20 10:07 ..
-rw-r--r--  1 sree sree   197 Oct 20 11:13 test.c
drwxr-xr-x  2 sree sree  4096 Oct 20 11:14 .
-rwxr-xr-x  1 sree sree 10840 Oct 20 11:14 test
sree@ubuntu:~/debugging$
```

gcc -std=c99 -g -o test test.C

Run GDB with the generated executable

Type the following command to start GDB with the compiled executable.

```
gdb ./test
```



```
sree@ubuntu:~/debugging$ gdb ./test
GNU gdb (Ubuntu 7.12.50.20170314-0ubuntu1) 7.12.50.20170314-git
Copyright (C) 2017 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 ./test...done.
(gdb) r
Starting program: /home/sree/debugging/test
-15808
[Inferior 1 (process 4989) exited normally]
(gdb) |
```

gdb ./test

Useful GDB commands:

Here are a few useful commands to get started with GDB.

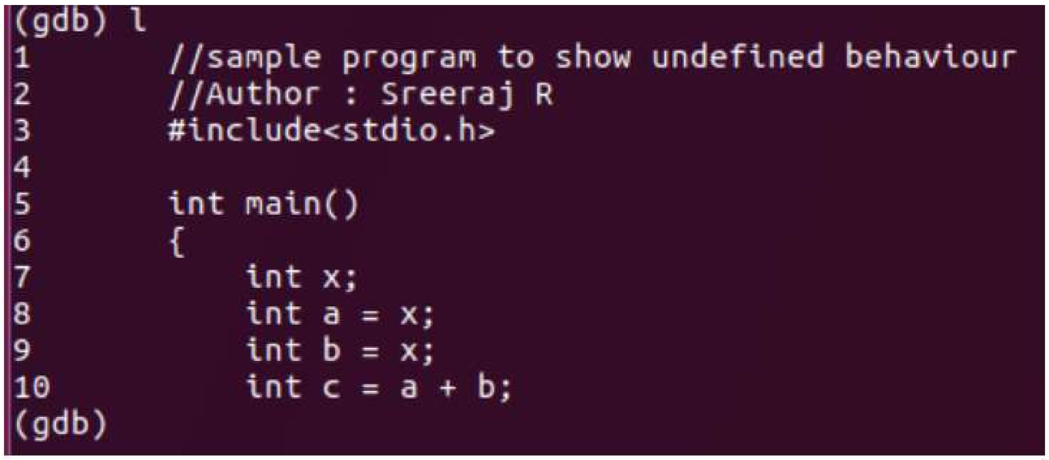
Here are a few useful commands to get started with GDB.

Command	Description
run or r	Executes the program from start to end.
break or b	Sets a breakpoint on a particular line.
disable	Disables a breakpoint
enable	Enables a disabled breakpoint.
next or n	Executes the next line of code without diving into functions.

Command	Description
step	Goes to the next instruction, diving into the function.
list or l	Displays the code.
print or p	Displays the value of a variable.
quit or q	Exits out of GDB.
clear	Clears all breakpoints.
continue	Continues normal execution

Display the code

Now, type "l" at gdb prompt to display the code.

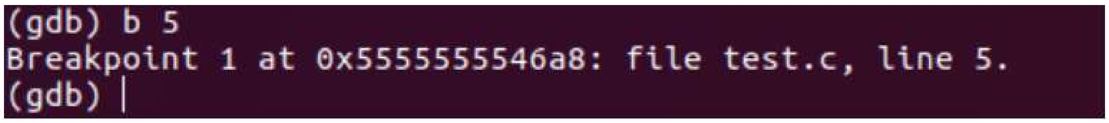


```
(gdb) l
1	//sample program to show undefined behaviour
2	//Author : Sreeraj R
3	#include<stdio.h>
4
5	int main()
6	{
7	    int x;
8	    int a = x;
9	    int b = x;
10	    int c = a + b;
(gdb)
```

Display the code

Set a breakpoint

Let's introduce a break point, say line 5.



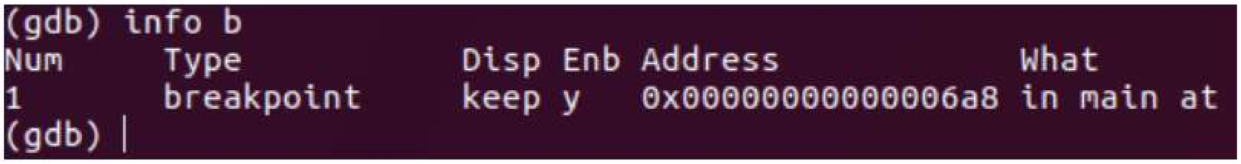
```
(gdb) b 5
Breakpoint 1 at 0x5555555546a8: file test.c, line 5.
(gdb) |
```

Set a breakpoint

If you want to put breakpoint at different lines, you can type "b *line_number*".By default "list or l" display only first 10 lines.

View breakpoints

In order to see the breakpoints, type "info b".

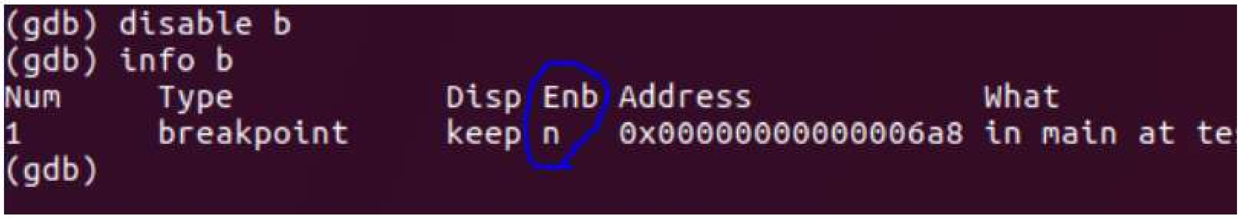


```
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000000006a8 in main at
(gdb) |
```

View breakpoints

Disable a breakpoint

Having done the above, let's say you changed your mind and you revert. Type "disable b".

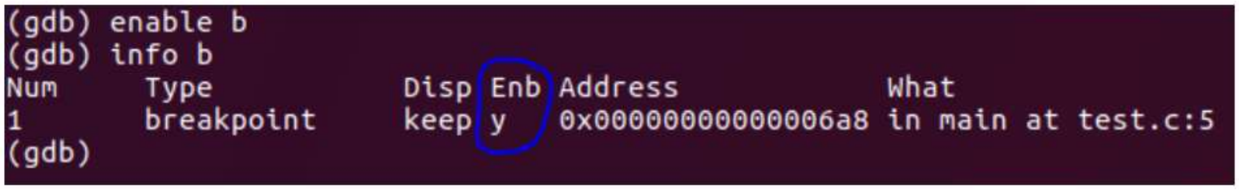


```
(gdb) disable b
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep n   0x00000000000006a8 in main at te
(gdb)
```

Disable a breakpoint

Re-enable a disabled breakpoint

As marked in the blue circle, Enb becomes n for disabled. 9. To re-enable the recent disabled breakpoint. Type "enable b".

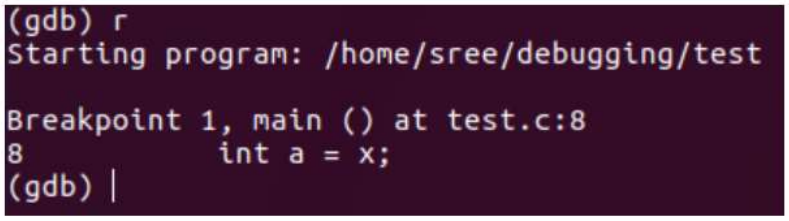


```
(gdb) enable b
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000000006a8 in main at test.c:5
(gdb)
```

Re-enable a disabled breakpoint

Run the code

Run the code by typing "run or r".If you haven't set any breakpoints, the run command will simply execute the full program.



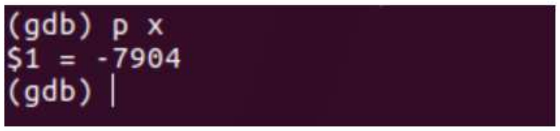
```
(gdb) r
Starting program: /home/sree/debugging/test

Breakpoint 1, main () at test.c:8
8           int a = x;
(gdb) |
```

Run the code

Print variable values

To see the value of variable, type "print *variable_name* or p *variable_name*".



```
(gdb) p x
$1 = -7904
(gdb) |
```

Print variable values

The above shows the values stored at x at time of execution.

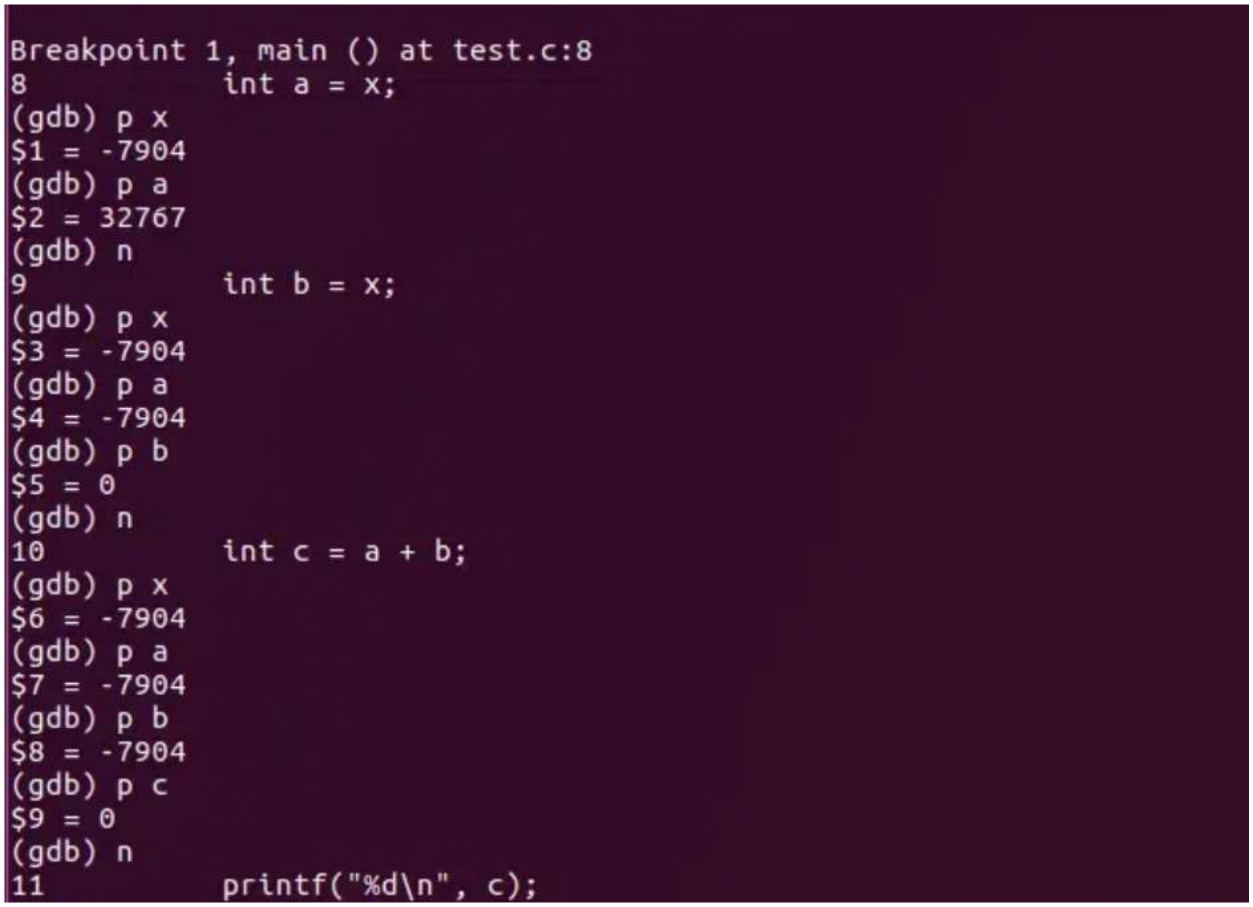
Change variable values

To change the value of variable in gdb and continue execution with changed value, type "set *variable_name*".

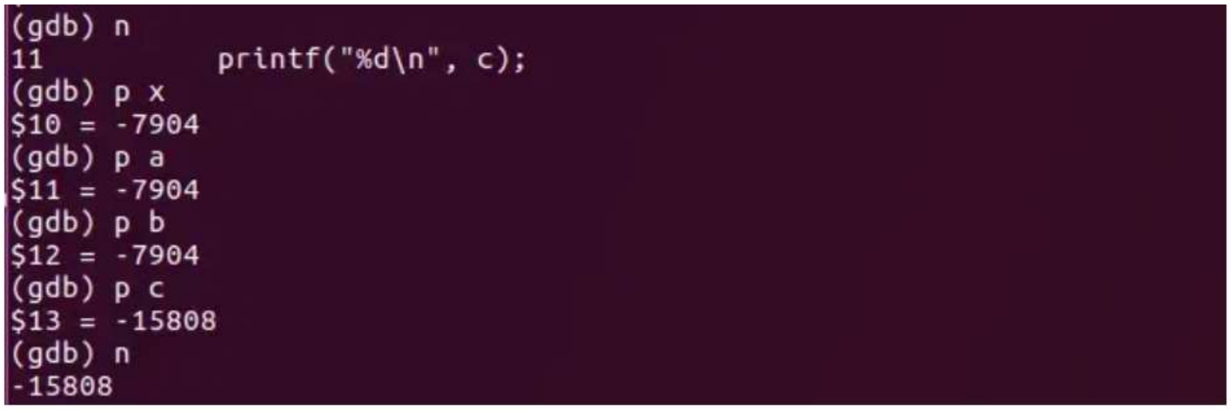
Debugging output

Below screenshot shows the values of variables from which it's quite understandable the reason why we got a garbage value as output. At every execution of **./test** we will be receiving a different output.

Exercise: Try using set x = 0 in gdb at first run and see the output of c.



```
Breakpoint 1, main () at test.c:8
8           int a = x;
(gdb) p x
$1 = -7904
(gdb) p a
$2 = 32767
(gdb) n
9           int b = x;
(gdb) p x
$3 = -7904
(gdb) p a
$4 = -7904
(gdb) p b
$5 = 0
(gdb) n
10          int c = a + b;
(gdb) p x
$6 = -7904
(gdb) p a
$7 = -7904
(gdb) p b
$8 = -7904
(gdb) p c
$9 = 0
(gdb) n
11          printf("%d\n", c);
```



```
(gdb) n
11              printf("%d\n", c);
(gdb) p x
$10 = -7904
(gdb) p a
$11 = -7904
(gdb) p b
$12 = -7904
(gdb) p c
$13 = -15808
(gdb) n
-15808
```

Debugging output

GDB offers many more ways to debug and understand your code like examining stack, memory, threads, manipulating the program, etc. I hope the above example helps you get started with gdb.

Conclusion

In this article we have discussed GDB (GNU Debugger) which is a powerful tool in Linux used for debugging C programs. We have discussed some of the following steps so that we can compile your code with debugging information, run GDB, set breakpoint, examine variables, and analyze program behavior. We have also discussed GDB's features, such as code examination, breakpoint management, variable manipulation, and program execution control which allow us to efficiently debug and issue resolution.