***Department of Computer Science***

***Parallel and Distributed Computing***

***GNU Debugger***

**Dated: 12/03/2021**

**Table of Contents**

[**What is GNU Debugger?** 3](#_Toc69198126)

[**What Languages does GDB Support?** 4](#_Toc69198127)

[**How GDB Debugs?** 4](#_Toc69198128)

[**GDB - Debugging Symbols** 4](#_Toc69198129)

[**GDB - Commands** 5](#_Toc69198130)

[Breakpoints or Watchpoints 6](#_Toc69198131)

[Setting Variables 7](#_Toc69198132)

[Backtrace and Changing Frames 7](#_Toc69198133)

[Handling Signals 8](#_Toc69198134)

[**Debugging Programs with Multiple Threads** 8](#_Toc69198135)

[**GDB Debugging Examples** 9](#_Toc69198136)

[Example#1 Debugging on Linux Machine 9](#_Toc69198137)

[**Code** 9](#_Toc69198138)

[**Debugging** 11](#_Toc69198139)

[Example#2 Online Debugging 21](#_Toc69198140)

[**Interface** 22](#_Toc69198141)

[**Code** 22](#_Toc69198142)

[**Output** 23](#_Toc69198143)

[**Debugging** 23](#_Toc69198144)

[**Saving a program in Online Debugger** 24](#_Toc69198145)

[Example#3 Online Debugging with PThreads 25](#_Toc69198146)

[**Background** 25](#_Toc69198147)

[**Code** 27](#_Toc69198148)

[**Explanation** 29](#_Toc69198149)

[**Debugging** 29](#_Toc69198150)

[**Other debuggers** 30](#_Toc69198151)

[DBX Debugger 30](#_Toc69198152)

[DDD Debugger 30](#_Toc69198153)

[Advantages of DDD Debugger over GDB 30](#_Toc69198154)

[**Conclusion** 31](#_Toc69198155)

[**References** 32](#_Toc69198156)

# **What is GNU Debugger?**

A debugger is a program that runs other programs, allowing the user to exercise control over these programs, and to examine variables when problems arise. Debuggers also allow programmers to run a program step by step so that they can determine exactly when and why a program crashes.

GNU Debugger, which is also called **gdb,** is the most popular debugger for UNIX systems to debug C and C++ programs. 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 compilation process.

GDB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed.

GNU Debugger helps you in getting information about the following:

* If a core dump happened, then what statement or expression did the program crash on?
* If an error occurs while executing a function, what line of the program contains the call to that function, and what are the parameters?
* What are the values of program variables at a particular point during execution of the program?
* What is the result of a particular expression in a program?

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

* Start your program, specifying anything that might affect its behavior.
* Make your program stop on specified conditions.
* Examine what has happened, when your program has stopped.
* Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

GNU or [GDB debugger](https://www.gnu.org/software/gdb/download) is an application for finding out how your C or C++ program runs or for analyzing the moment the program crashes. You can perform many useful tasks with GDB: run the program, stop the program under specific conditions, analyze the situation, make modifications, and test new changes.

# **What Languages does GDB Support?**

GDB supports the following languages (in alphabetical order):

* Ada
* Assembly
* C
* C++
* D
* Fortran
* Go
* Objective-C
* OpenCL
* Modula-2
* Pascal
* Rust

# **How GDB Debugs?**

GDB allows you to run the program up to a certain point, then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.

GDB uses a simple command line interface.

# **GDB - Debugging Symbols**

A **Debugging Symbol Table** maps instructions in the compiled binary program to their corresponding variable, function, or line in the source code. This mapping could be something like:

* Program instruction ⇒ item name, item type, original file, line number defined.

# **GDB - Commands**

GDB offers a big list of commands, however the following commands are the ones used most frequently:

* **r** - Runs the program until a breakpoint or error
* **c** - Continues running the program until the next breakpoint or error
* **finish**
  + Finishes executing the current function, then pause (also called step out). Useful if you accidentally stepped into a function.
* **step**
  + Runs the next instruction, not line. If the current instruction is setting a variable, it is the same as **next**. If it’s a function, it will jump into the function, execute the first statement, then pause. **step** is good for diving into the details of your code.
* **next**
  + Runs the program until next line, then pauses. If the current line is a function, it executes the entire function, then pauses. **next** is good for walking through your code quickly.
* **Until**

until is like next, except that if you are at the end of a loop, until will continue execution until the loop is exited, whereas next will just take you back up to the beginning of the loop. This is convenient if you want to see what happens after the loop, but don't want to step through every iteration

* **s N** - Runs the next N lines of the program
* **u N** - Runs until you get N lines in front of the current line
* **p var** - Prints the current value of the variable "var"
* **bt** - Prints a stack trace
* **u** - Goes up a level in the stack
* **d** - Goes down a level in the stack
* **q** - Quits gdb
* Step commands:

By using step commands one can execute program line by line

• continue – Resume program execution until next breakpoint is reached.

• step into – Execute program line by line stepping into function.

• step over – Execute program line by line but don’t go inside function call.

• step out – Resume program execution until current function is finished.

## Breakpoints or Watchpoints

* **b main** - Puts a breakpoint at the beginning of the program
* **b** - Puts a breakpoint at the current line
* **b N** - Puts a breakpoint at line N
* **b +N** - Puts a breakpoint N lines down from the current line
* **b fn** - Puts a breakpoint at the beginning of function "fn"
* **continue**
  + Resumes execution after being paused by a breakpoint/watchpoint. The program will continue until it hits the next breakpoint/watchpoint.
* **delete N**
  + Deletes breakpoint N (breakpoints are numbered when created).
* **info break** - list breakpoints

Breakpoints play an important role in debugging. They pause (break) a program when it reaches a certain point. You can examine and change variables and resume execution. This is helpful when some input failure occurs, or inputs are to be tested.

* break 45
* break myfunction
  + Sets a breakpoint at line 45, or at myfunction. The program will pause when it reaches the breakpoint.
* watch x == 3
  + Sets a watchpoint, which pauses the program when a condition changes (when x == 3 changes). Watchpoints are great for certain inputs (myPtr != NULL) without having to break on *every* function call.

## Setting Variables

Viewing and changing variables at runtime is a critical part of debugging. Try providing invalid inputs to functions or running other test cases to find the root cause of problems. Typically, you will view/set variables when the program is paused.

* print x
  + Prints current value of variable x. Being able to use the original variable names is why the (-g) flag is needed; programs compiled regularly have this information removed.
* set x = 3
* set x = y
  + Sets x to a set value (3) or to another variable (y)
* call myfunction()
* call myotherfunction(x)
* call strlen(mystring)
  + Calls user-defined or system functions. This is extremely useful, but beware of calling buggy functions.
* display x
  + Constantly displays the value of variable x, which is shown after every step or pause. Useful if you are constantly checking for a certain value.
* undisplay x
  + Removes the constant display of a variable displayed by display command.

## Backtrace and Changing Frames

A stack is a list of the current function calls - it shows you where you are in the program. A *frame* stores the details of a single function call, such as the arguments.

* **bt**
  + **Backtraces** or prints the current function stack to show where you are in the current program. If main calls function a(), which calls b(), which calls c(), the backtrace is
* **down**
  + Move to the next frame up or down in the function stack. If you are in **c,** you can move to **b** or **a** to examine local variables.
* **return**
  + Returns from current function.

## Handling Signals

Signals are messages thrown after certain events, such as a timer or error. GDB may pause when it encounters a signal; you may wish to ignore them instead.

* handle [signalname] [action]
* handle SIGUSR1 nostop
* handle SIGUSR1 noprint
* handle SIGUSR1 ignore
  + Instruct GDB to ignore a certain signal (SIGUSR1) when it occurs. There are varying levels of ignoring [1].

# **Debugging Programs with Multiple Threads**

In some operating systems, such as GNU/Linux and Solaris, a single program may have more than one thread of execution. The precise semantics of threads differ from one operating system to another, but in general the threads of a single program are akin to multiple processes—except that they share one address space (that is, they can all examine and modify the same variables). On the other hand, each thread has its own registers and execution stack, and perhaps private memory.

GDB provides these facilities for debugging multi-thread programs:

* automatic notification of new threads
* ‘thread *thread-id*’, a command to switch among threads
* ‘info threads’, a command to inquire about existing threads
* ‘thread apply [*thread-id-list* | all] *args*’, a command to apply a command to a list of threads
* thread-specific breakpoints
* ‘set print thread-events’, which controls printing of messages on thread start and exit.
* ‘set libthread-db-search-path *path*’, which lets the user specify which libthread\_db to use if the default choice isn’t compatible with the program.

The GDB thread debugging facility allows you to observe all threads while your program runs—but whenever GDB takes control, one thread in particular is always the focus of debugging. This thread is called the current thread. Debugging commands show program information from the perspective of the current thread.

Whenever GDB detects a new thread in your program, it displays the target system’s identification for the thread with a message in the form ‘[New *systag*]’, where *systag* is a thread identifier whose form varies depending on the particular system. For example, on GNU/Linux, you might see

[New Thread 0x41e02940 (LWP 25582)]

when GDB notices a new thread. In contrast, on other systems, the *systag* is simply something like ‘process 368’, with no further qualifier. For debugging purposes, GDB associates its own thread number —always a single integer.

From GDB’s perspective, a process always has at least one thread. In other words, GDB assigns a thread number to the program’s “main thread” even if the program is not multi-threaded. If GDB detects the program is multi-threaded, it augments the usual message about stopping at a breakpoint with the ID and name of the thread that hit the breakpoint.

Thread 2 "client" hit Breakpoint 1, send\_message () at client.c:68

Likewise when the program receives a signal:

Thread 1 "main" received signal SIGINT, Interrupt.

info threads [*thread-id-list*]

Display information about one or more threads. With no arguments displays information about all threads. You can specify the list of threads that you want to display using the thread ID list syntax [2].

# **GDB Debugging Examples**

GDB can work on different operating systems like UNIX, Linux and Solaris etc. Following are some examples of GDB debugging in different systems.

## Example#1 Debugging on Linux Machine

### **Code**

The program to be debugged should be compiled with **-g** option. The below given C++ file that is saved as **gfg.cpp**. We are going to use this file in this article.

|  |
| --- |
| #include <iostream>  #include <stdlib.h>  #include <string.h>  using namespace std;    int findSquare(int a)  {      return a \* a;  }    int main(int n, char\*\* args)  {      for (int i = 1; i < n; i++)      {          int a = atoi(args[i]);          cout << findSquare(a) << endl;      }      return 0;  } |

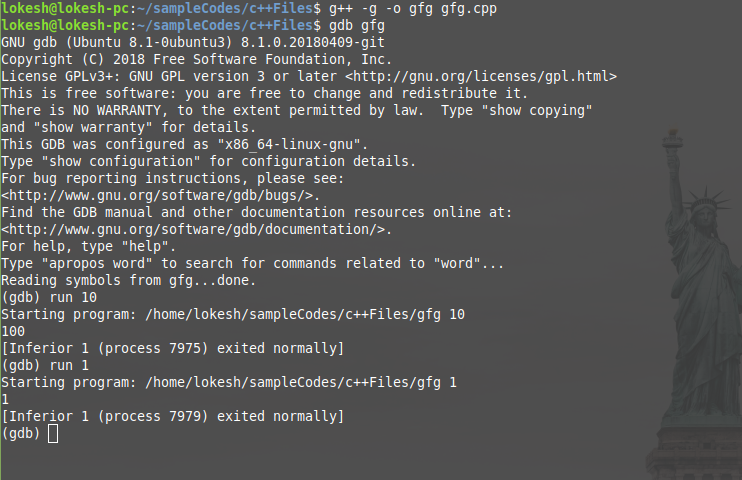
Compile the above C++ program using the command:

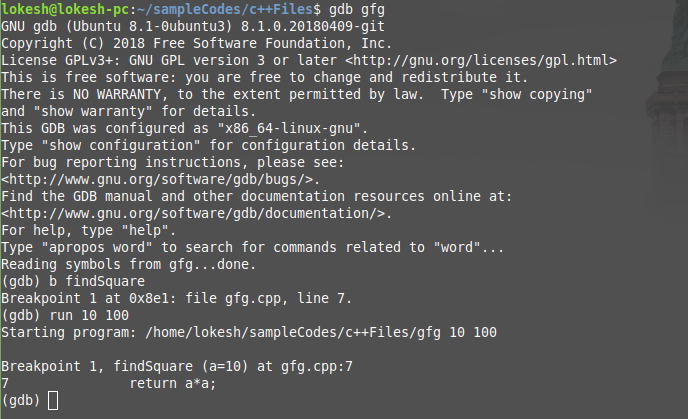
g++ -g -o gfg gfg.cpp

### **Debugging**

To start the debugger of the above **gfg** executable file, enter the command **gdb gfg**. It opens the gdb console of the current program, after printing the version information.

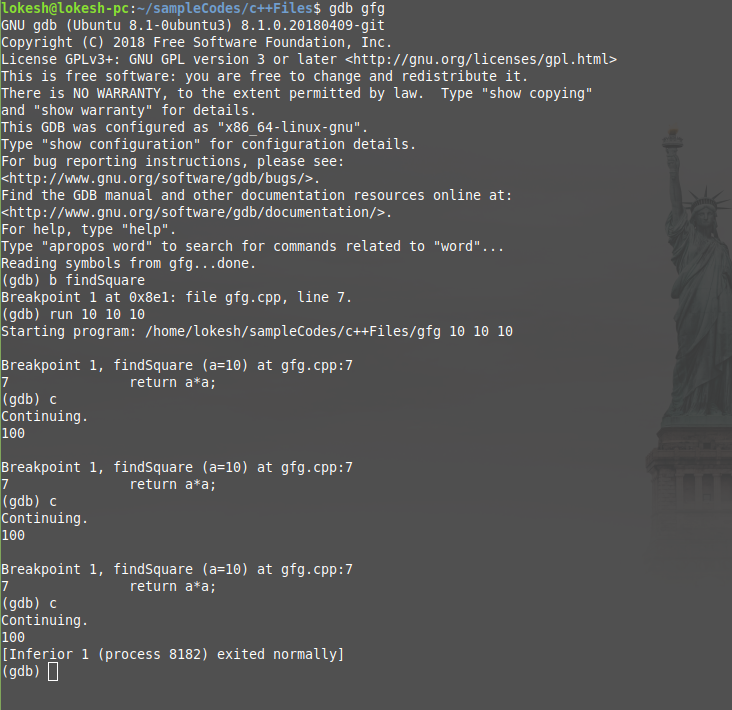
1. **run [args] :**This command runs the current executable file. In the below image, the program was executed twice, one with the command line argument 10 and another with the command line argument 1, and their corresponding outputs were printed.



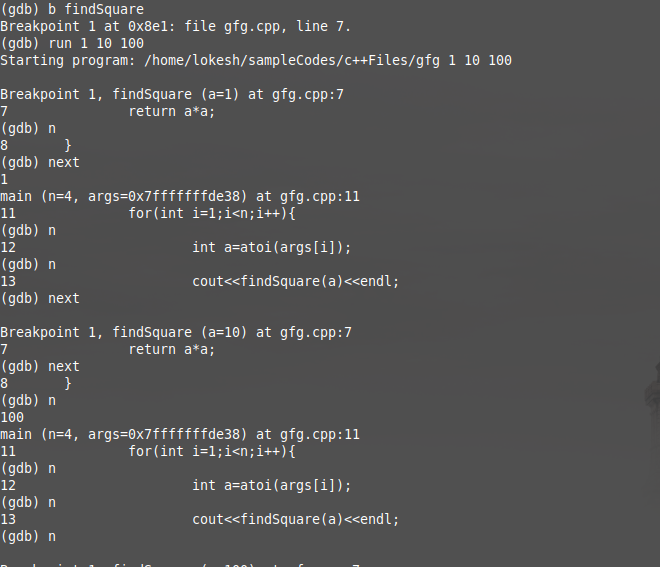
1. **quit or q :** To quit the gdb console, either **quit** or **q** can be used.
2. **help :** It launches the manual of gdb along with all list of classes of individual commands.
3. **break :** The command **break [function name]** helps to pause the program during execution when it starts to execute the function. It helps to debug the program at that point. Multiple breakpoints can be inserted by executing the command wherever necessary. **b findSquare** command makes the gfg executable pause when the debugger starts to execute the findSquare function.
4. b
5. break [function name]
6. break [file name]:[line number]
7. break [line number]
8. break \*[address]
9. break \*\*\*any of the above arguments\*\*\* if [condition]
10. b \*\*\*any of the above arguments\*\*\*

In the above example, the program that was being executed(**run 10 100**), paused when it encountered findSquare function call. The program pauses whenever the function is called. Once the command is successful, it prints the breakpoint number, information of the program counter, file name, and the line number. As it encounters any breakpoint during execution, it prints the breakpoint number, function name with the values of the arguments, file name, and line number. The breakpoint can be set either with the address of the instruction (in hexadecimal form preceded with \*0x) or the line number and it can be combined with if condition (if the condition fails, the breakpoint will not be set) For example, **break findSquare if a == 10**.

1. **Continue:** This command helps to resume the current executable after it is paused by the breakpoint. It executes the program until it encounters any breakpoint or runs time error or the end of the program.
2. continue [repeat count]
3. c [repeat count]

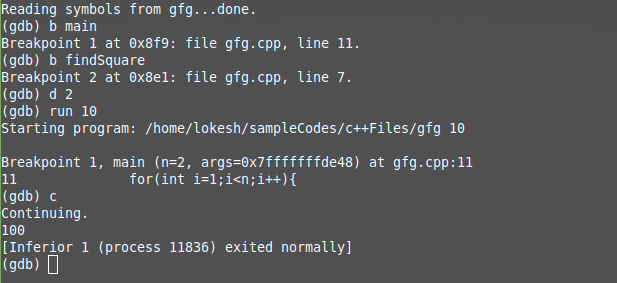


1. **next or n :** This command helps to execute the next instruction after it encounters the breakpoint.



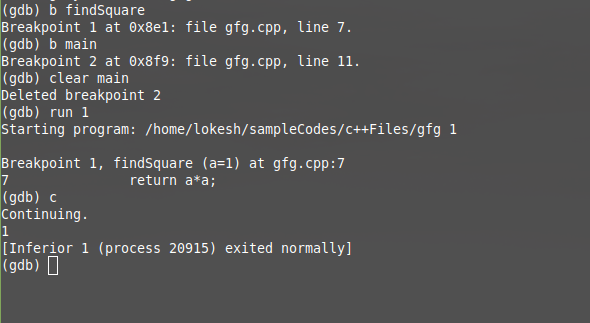
Whenever it encounters the above command, it executes the next instruction of the executable by printing the line in execution.

1. **delete :** This command helps to deletes the breakpoints and checkpoints. If the delete command is executed without any arguments, it deletes all the breakpoints without modifying any of the checkpoints. Similarly, if the checkpoint of the parent process is deleted, all the child checkpoints are automatically deleted.
2. d
3. delete
4. delete [breakpoint number 1] [breakpoint number 2] ...
5. delete checkpoint [checkpoint number 1] [checkpoint number 2] ...



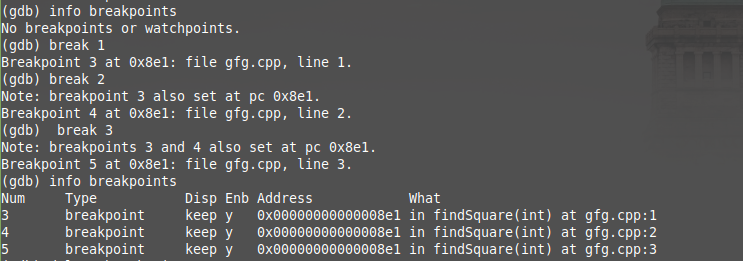
In the above example, two breakpoints were defined, one at the main and the other at the findSquare. Using the above command findSquare breakpoint was deleted. If there is no argument after the command, the command deletes all the breakpoints.

1. **clear :** This command deletes the breakpoint which is at a particular function with the name FUNCTION\_NAME. If the argument is a number, then it deletes the breakpoint that lies in that particular line.
2. clear [line number]
3. clear [FUNCTION\_NAME]

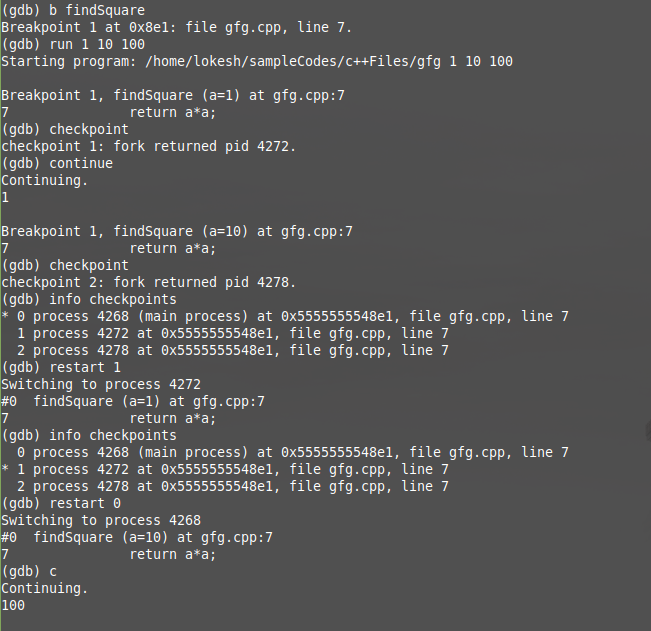


In the above example, once the clear command is executed, the breakpoint is deleted after printing the breakpoint number.

1. **disable [breakpoint number 1] [breakpoint number 2] …. :** Instead of deleting or clearing the breakpoints, they can be disabled and can be enabled whenever they are necessary.
2. **enable [breakpoint number 1] [breakpoint number 2] …. :** To enable the disabled breakpoints, this command is used.
3. **info :** When the info breakpoints in invoked, the breakpoint number, type, display, status, address, the location will be displayed. If the breakpoint number is specified, only the information about that particular breakpoint will be displayed. Similarly, when the info checkpoints are invoked, the checkpoint number, the process id, program counter, file name, and line number are displayed.
4. info breakpoints [breakpoint number 1] [breakpoint number 2] ...
5. info checkpoints [checkpoint number 1] [checkpoint number 2] ...

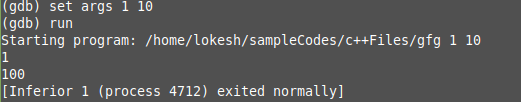


1. ***checkpoint*command and *restart*command :**These command creates a new process and keep that process in the suspended mode and prints the created process’s process id.

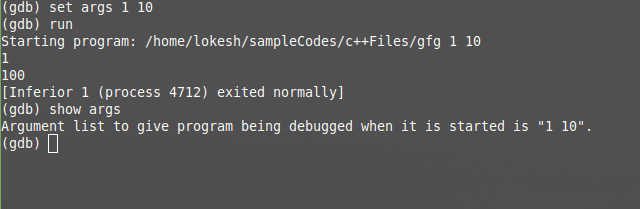


**For example**, in the above execution, the breakpoint is kept at function *findSquare*and the program was executed with the arguments “1 10 100”. When the function is called initially with *a = 1*, the breakpoint happens. Now we create a checkpoint and hence gdb returns a process id(4272), keeps it in the suspended mode and resumes the original thread once the continue command is invoked. Now the breakpoint happens with *a = 10* and another checkpoint(pid = 4278) is created. From the info checkpoint information, the asterisk mentions the process that will run if the gdb encounters a continue. To resume a specific process, **restart** command is used with the argument that specifies the serial number of the process. If all the process are finished executing, the **info checkpoint** command returns nothing.

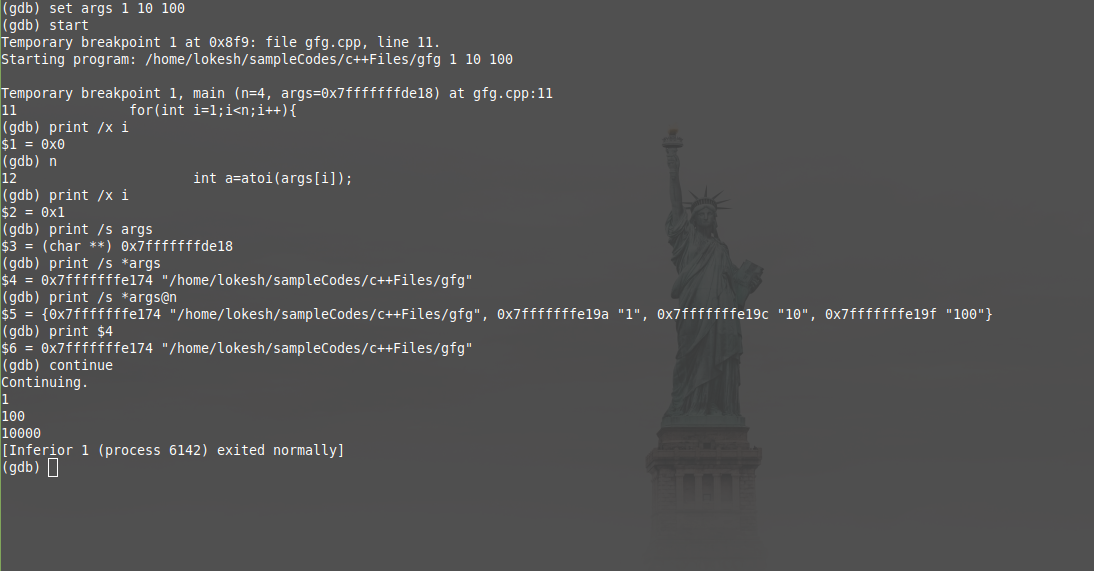
1. **set args [arg1] [arg2] … :** This command creates the argument list and it passes the specified arguments as the command line arguments whenever the **run** command without any argument is invoked. If the **run** command is executed with arguments after **set args**, the arguments are updated. Whenever the **run** command is ran without the arguments, the arguments are set by default.



1. **show args :** The show args prints the default arguments that will passed if the **run** command is executed. If either **set args** or **run** command is executed with the arguments, the default arguments will get updated, and can be viewed using the above **show args** command.



1. **print :** This command prints the value of a given expression. The display command prints all the previously displayed values whenever it encounters a breakpoint or the next command, whereas the print command saves all the previously displayed values and prints whenever it is called.
2. print [Expression]
3. print $[Previous value number]
4. print {[Type]}[Address]
5. print [First element]@[Element count]
6. print /[Format] [Expression]

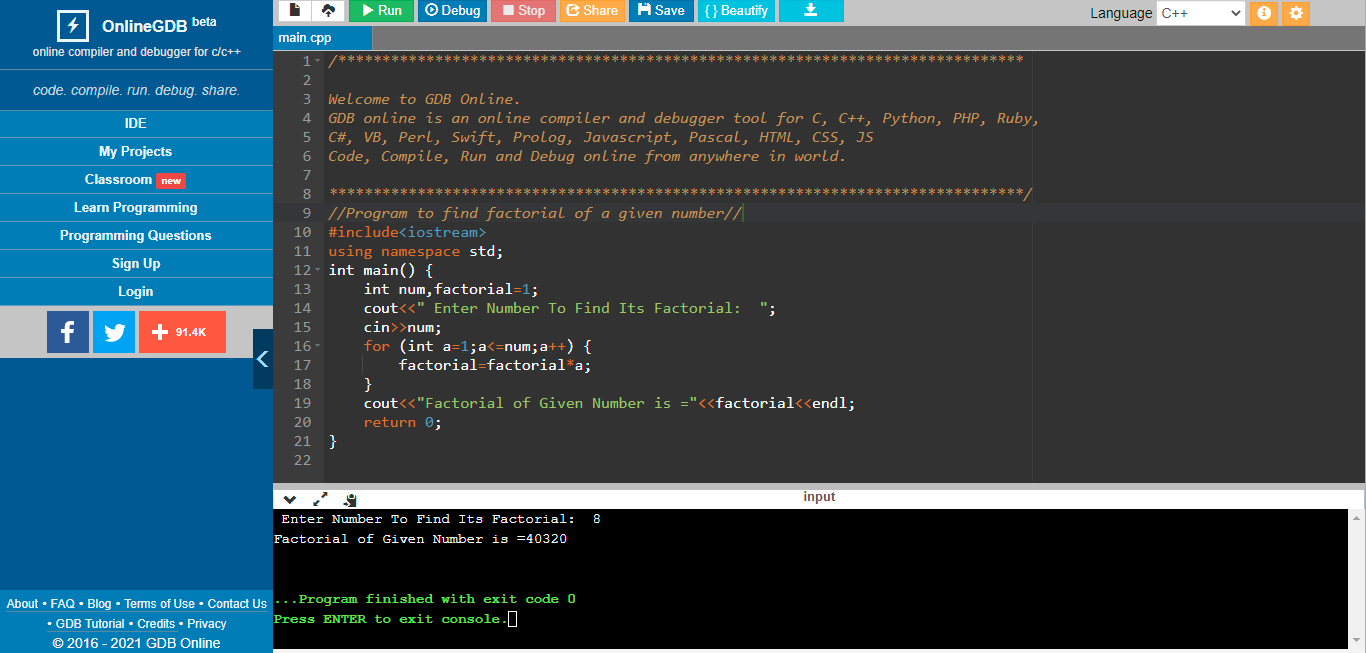


1. **file :** gdb console can be opened using the command **gdb** command. To debug the executables from the console, **file [executable filename]** command is used [3].

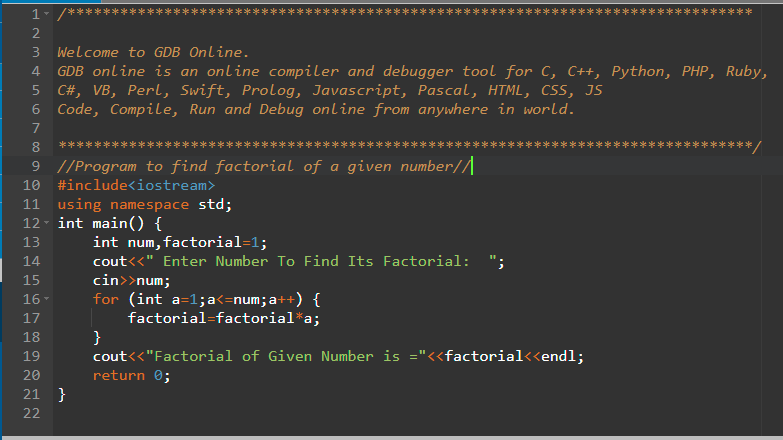
## Example#2 Online Debugging

<https://www.onlinegdb.com/>

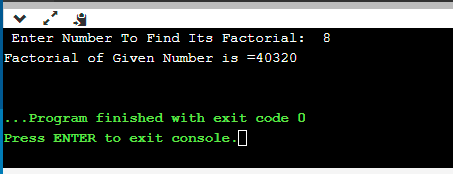
### **Interface**



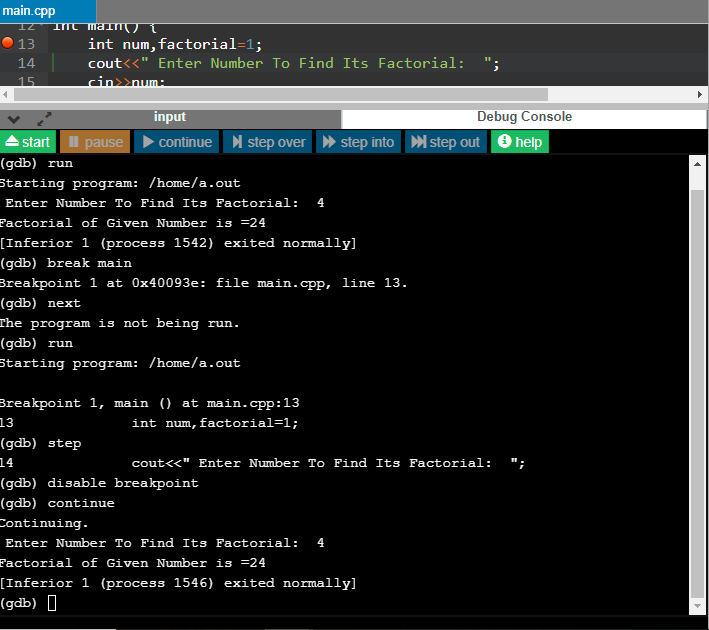
### **Code**

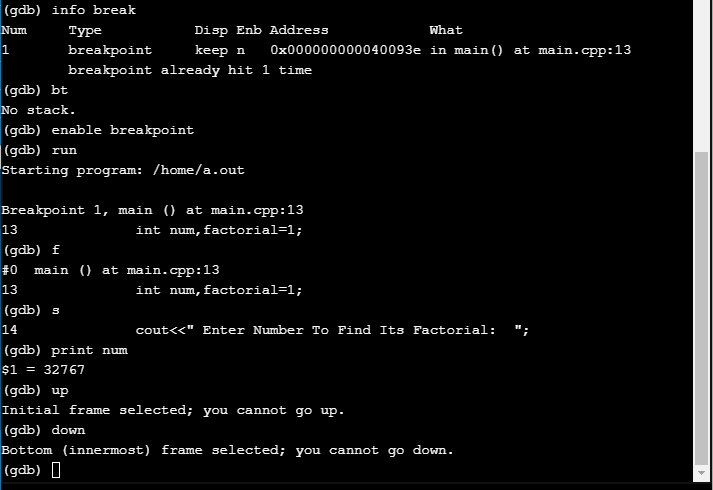


### **Output**



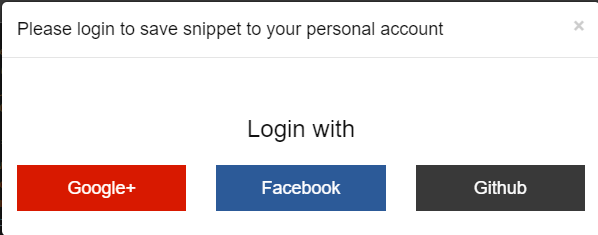
### **Debugging**

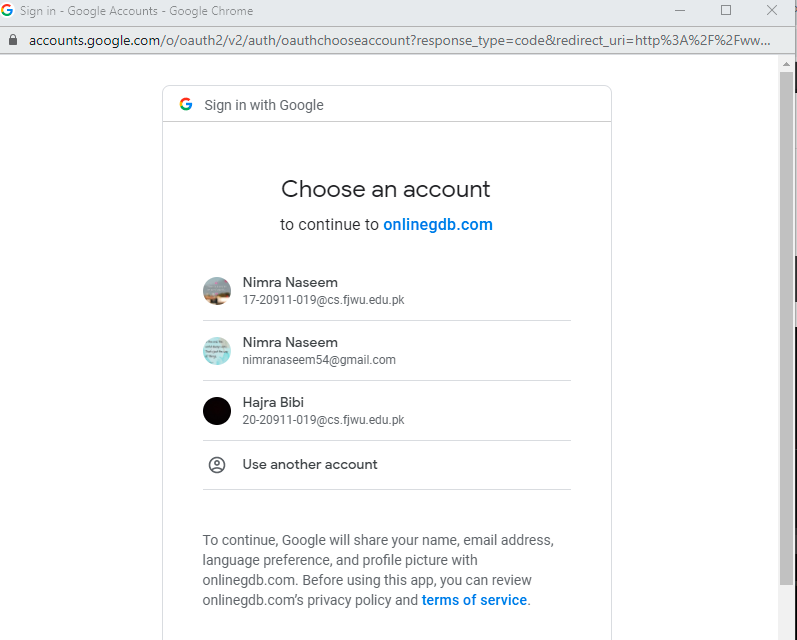


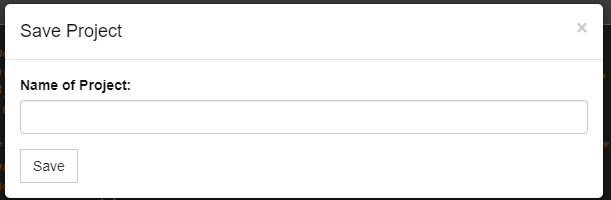


### **Saving a program in Online Debugger**









## Example#3 Online Debugging with PThreads

### **Background**

In a **Unix/Linux operating system**, the **C/C++ languages** provide the [POSIX thread (pthread)](https://www.geeksforgeeks.org/multithreading-c-2/) standard API (Application program Interface) for all thread related functions. It allows us to create multiple threads for concurrent process flow. It is most effective on multiprocessor or multi-core systems where threads can be implemented on a kernel-level for achieving the speed of execution. We must include the pthread.h header file at the beginning of the script to use all the functions of the pthreads library.

**pthread\_create:** used to create a new thread

**Syntax:**

int pthread\_create(pthread\_t \* thread,

const pthread\_attr\_t \* attr,

void \* (\*start\_routine)(void \*),

void \*arg);

**Parameters:**

* **thread:** pointer to an unsigned integer value that returns the thread id of the thread created.
* **attr:** pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
* **start\_routine:** pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void \*. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
* **arg:** pointer to void that contains the arguments to the function defined in the earlier argument

**pthread\_exit:** used to terminate a thread

**Syntax:**

void pthread\_exit(void \*retval);

**Parameters:** This method accepts a mandatory parameter **retval** which is the pointer to an integer that stores the return status of the thread terminated. The scope of this variable must be global so that any thread waiting to join this thread may read the return status.

***pthread\_join:*** used to wait for the termination of a thread.

**Syntax:**

int pthread\_join(pthread\_t th,

void \*\*thread\_return);

**Parameter:** This method accepts following parameters:

* + **th:** thread id of the thread for which the current thread waits.
  + **thread\_return:** pointer to the location where the exit status of the thread mentioned in th is stored.

***pthread\_self:*** used to get the thread id of the current thread.

**Syntax:**

pthread\_t pthread\_self(void);

***pthread\_equal:*** compares whether two threads are the same or not. If the two threads are equal, the function returns a non-zero value otherwise zero.

**Syntax:**

int pthread\_equal(pthread\_t t1,

pthread\_t t2);

**Parameters:** This method accepts following parameters:

* + t1: the thread id of the first thread
  + t2: the thread id of the second thread

***pthread\_cancel:*** used to send a cancellation request to a thread

**Syntax:**

int pthread\_cancel(pthread\_t thread);

**Parameter:** This method accepts a mandatory parameter **thread** which is the thread id of the thread to which cancel request is sent.

***pthread\_detach:*** used to detach a thread. A detached thread does not require a thread to join on terminating. The resources of the thread are automatically released after terminating if the thread is detached.

**Syntax:**

int pthread\_detach(pthread\_t thread);

**Parameter:** This method accepts a mandatory parameter **thread** which is the thread id of the thread that must be detached.

### **Code**

// C program to show thread functions

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

void\* func(void\* arg)

{

// detach the current thread

// from the calling thread

pthread\_detach(pthread\_self());

printf("Inside the thread\n");

// exit the current thread

pthread\_exit(NULL);

}

void fun()

{

pthread\_t ptid;

// Creating a new thread

pthread\_create(&ptid, NULL, &func, NULL);

printf("This line may be printed"

" before thread terminates\n");

// The following line terminates

// the thread manually

// pthread\_cancel(ptid);

// Compare the two threads created

if(pthread\_equal(ptid, pthread\_self()))

printf("Threads are equal\n");

else

printf("Threads are not equal\n");

// Waiting for the created thread to terminate

pthread\_join(ptid, NULL);

printf("This line will be printed"

" after thread ends\n");

pthread\_exit(NULL);

}

// Driver code

int main()

{

fun();

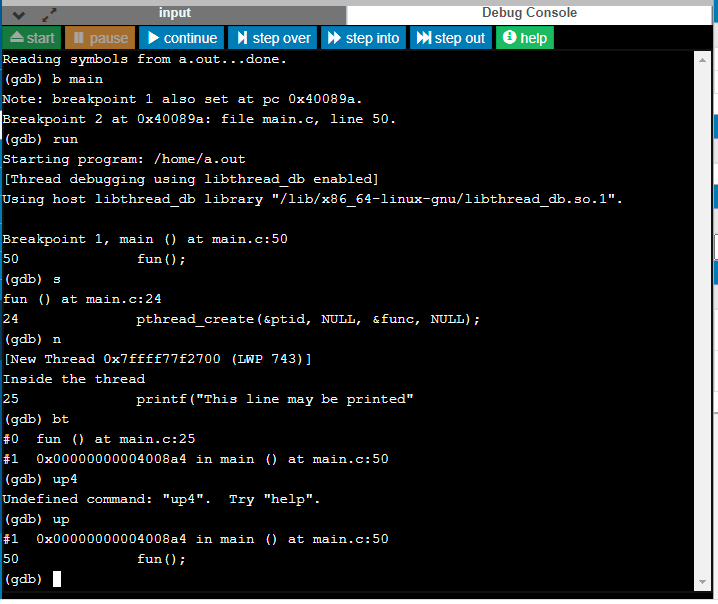
return 0;

}

### **Explanation**

 Here two threads of execution are created in the code. The order of the lines of output of the two threads may be interchanged depending upon the thread processed earlier. The main thread waits on the newly created thread for exiting. Therefore, the final line of the output is printed only after the new thread exits. The threads can terminate independently of each other by not using the pthread\_join function. If we want to terminate the new thread manually, we may use pthread\_cancel to do it [4].

### **Debugging**



# **Other debuggers**

There are quite a few good debuggers available in the market:

* **DBX Debugger** - This debugger ships along with Sun Solaris and you can get complete information about this debugger using the man page of dbx, i.e., *man dbx*.
* **DDD Debugger** - This is a graphical version of dbx and freely available on Linux. To have a complete detail, use the man page of ddd, i.e., *man ddd [1]*.

## Advantages of DDD Debugger over GDB

* The GUI interfaces provided by DDD and is more visually appealing than that of GDB. They also tend to be more convenient.
* In DDD your task would be far simpler: To clear a breakpoint, simply click the stop sign at the desired line, then click Clear, and the stop sign would disappear, showing that the breakpoint has been cleared.
* One task for which the GUIs are clear winners is stepping through code. It is much easier and more pleasant to do this using DDD rather than GDB, because you can watch your movement through the code in the GUI source code window. The next line in your source code to be executed is indicated by an arrow [5].

# **Conclusion**

GDB, the GNU Debugger, was among the first programs to be written for the Free Software Foundation, and it has been a staple of free and open-source software systems ever since. The basic theory is that GDB will replace a program instruction with a trap, illegal divide, or some other instruction that will cause an exception, and then when it's encountered, GDB will take the exception and stop the program. Even though GDB can help you in finding out memory leakage related bugs, but it is not a tool to detect memory leakages. GDB cannot be used for programs that compile with errors and it does not help in fixing those errors.

# 

# **References**

[1] <https://www.tutorialspoint.com/gnu_debugger/gdb_quick_guide.html>

[2] <https://sourceware.org/gdb/onlinedocs/gdb/Threads.html>

[3] <https://www.geeksforgeeks.org/gdb-command-in-linux-with-examples/>

[4] <https://www.geeksforgeeks.org/thread-functions-in-c-c/>

[5] <https://dl.acm.org/doi/10.5555/1457534>

[6] <https://www.onlinegdb.com/>