***GDB “GNU Debugger”***

“GNU Debugger” A debugger for several languages, including C and C++ It allows you to inspect what the program is doing at a certain point during execution. Errors like segmentation faults may be easier to find with the help of gdb.

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.

Normally we will compile the program:

gcc <flags> <source\_file> -o <output\_file>

gcc -Wall -g test.c -o test

Now you add a **-g** option to enable built-in debugging support (which gdb needs):

gcc [other flags] -g <source\_file> -o <output\_file>

gcc -Wall -g test.c -o test

Just try “**gdb**” or “**gdb test.x**” You’ll get a prompt that looks like this:

(gdb)

If you didn’t specify a program to debug, you’ll have to load it in now:

(gdb) file test

Here, ***test*** is the program you want to load, and “file” is the command to load it.

If ever confused about a command or just want more information, use the “help” command, with or without an argument:

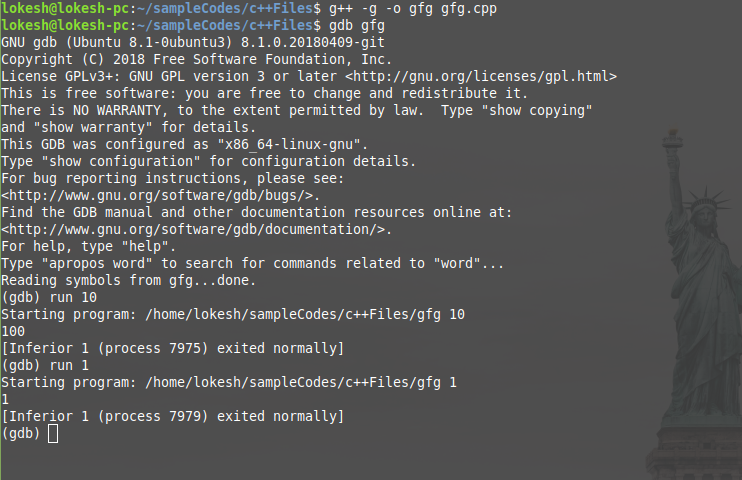
(gdb) help [command]

**gdb commands:**

**help** : It launches the manual of gdb along with all list of classes of individual commands.

**run** or r –> executes the program from start to end.  
**break** or b –> sets breakpoint on a particular line/function.  
**disable** -> disable a breakpoint.  
**enable** –> enable a disabled breakpoint.  
**next** or n -> executes next line of code, but don’t dive into functions.  
**step** –> go to next instruction, diving into the function.  
**list** or l –> displays the code.  
**print** or p –> used to display the stored value.  
**quit** or q –> exits out of gdb.  
**clear** –> to clear all breakpoints.  
**continue** –> continue normal execution.

**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.



**quit** or **q** : To quit the gdb console, either **quit** or **q** can be used.

**help** : It launches the manual of gdb along with all list of classes of individual commands.

**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.

b

break [function name]

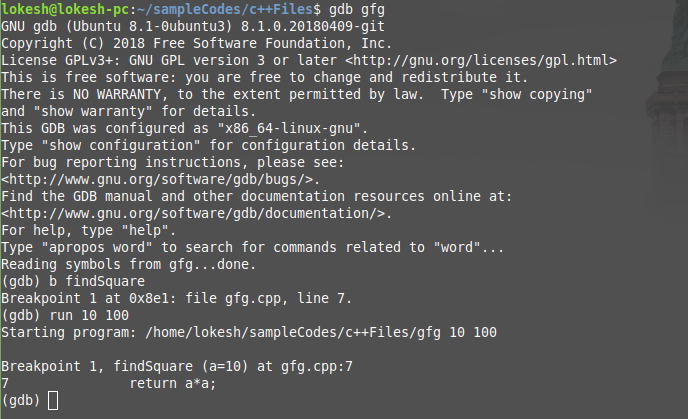
break [file name]:[line number]

break [line number]

break \*[address]

break \*\*\*any of the above arguments\*\*\* if [condition]

b \*\*\*any of the above arguments\*\*\*

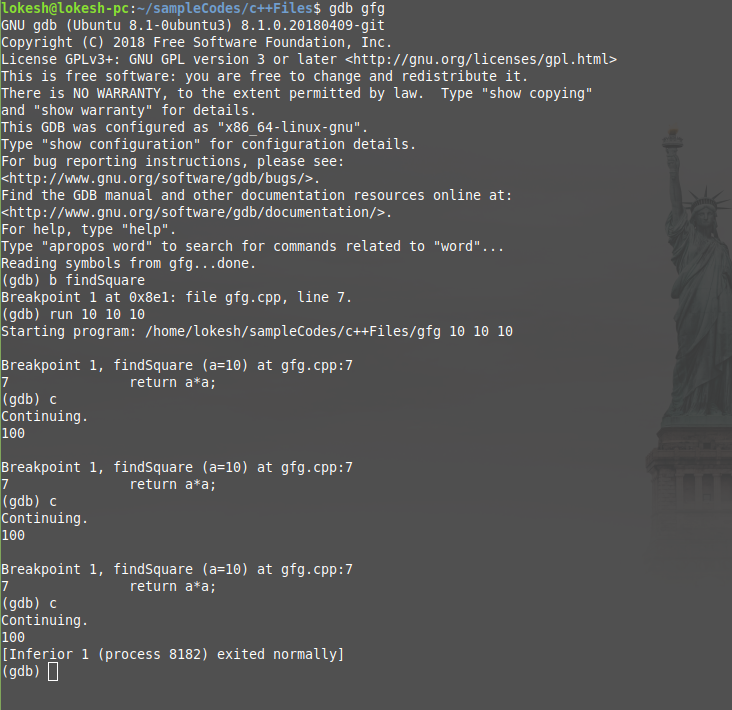


In the above example, the program that was being executed(run 10 100), paused when it encountered findSquare function call. The program pauses whenever the function is called. Once the command is successful, it prints the breakpoint number, information of the program counter, file name, and the line 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**.

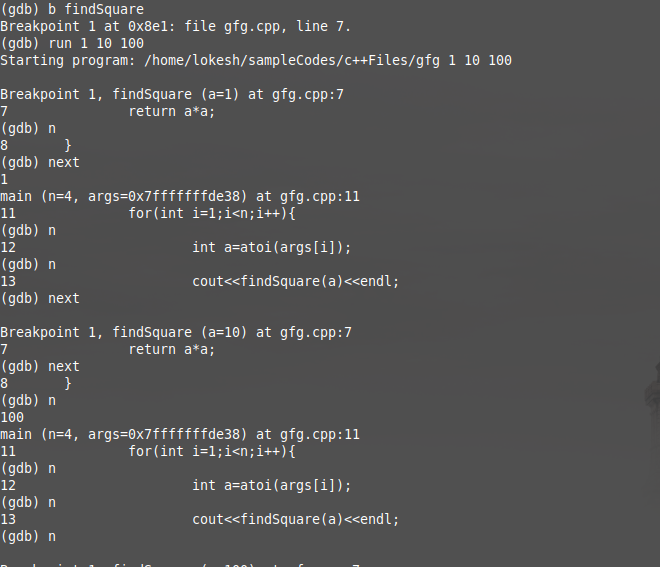
**continue** : This command helps to resume the current executable after it is paused by the breakpoint. It executes the program until it encounters any breakpoint or runs time error or the end of the program. If there is an integer in the argument(repeat count), it will consider it as the continue repeat count and will execute continue command “repeat count” number of times.

continue [repeat count]

c [repeat count]



**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.

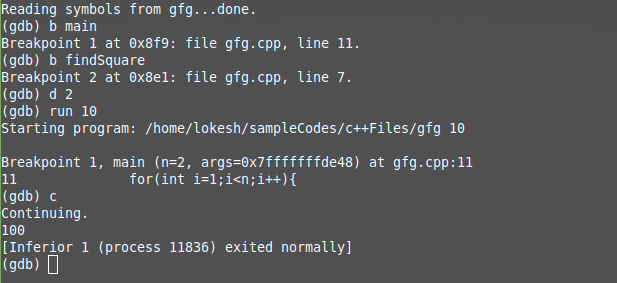
**delete** : This command helps to deletes the breakpoints and checkpoints. If the delete command is executed without any arguments, it deletes all the breakpoints without modifying any of the checkpoints. Similarly, if the checkpoint of the parent process is deleted, all the child checkpoints are automatically deleted.

d

delete

delete [breakpoint number 1] [breakpoint number 2] ...

delete checkpoint [checkpoint number 1] [checkpoint number 2] ...

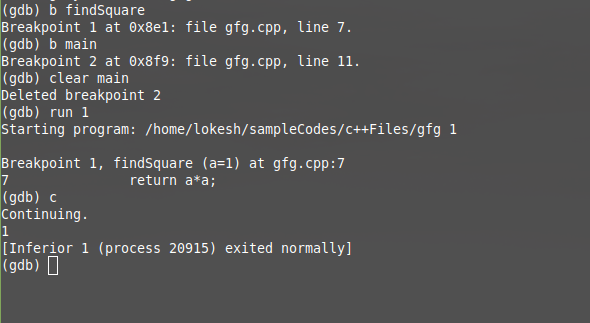


In the above example, two breakpoints were defined, one at the main and the other at the findSquare. Using the above command findSquare breakpoint was deleted. If there is no argument after the command, the command deletes all the breakpoints.

**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.

clear [line number]

clear [FUNCTION\_NAME]



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

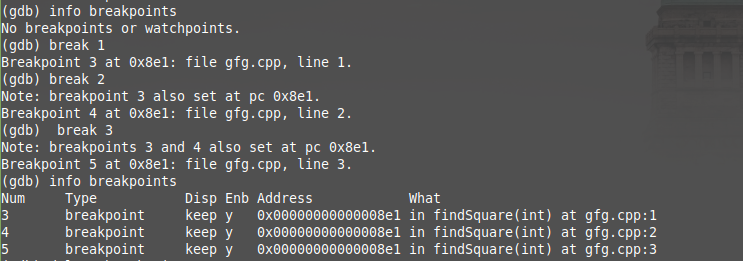
**disable** [breakpoint number 1] [breakpoint number 2] …. : Instead of deleting or clearing the breakpoints, they can be disabled and can be enabled whenever they are necessary.

**enable** [breakpoint number 1] [breakpoint number 2] …. : To enable the disabled breakpoints, this command is used.

**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.

info breakpoints [breakpoint number 1] [breakpoint number 2] ...

info checkpoints [checkpoint number 1] [checkpoint number 2] ...



## **Printing source lines:**

## To print lines from a source file, use the **list** command (abbreviated **l**). By default, ten lines are printed. There are several ways to specify what part of the file you want to print.

Here are the forms of the **list** command most commonly used:

list linenum

Prints lines centered around line number **linenum** in the current source file.

list function

Prints lines centered around the beginning of function **function**.

list

Prints more lines. If the last lines printed were printed with a **list** command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame, this prints lines centered around that line.

list -

Print lines just before the lines last printed.

By default, GDB prints ten source lines with any of these forms of the **list** command. You can change this using set listsize:

set listsize count

Make the **list** command display **count** source lines (unless the **list** argument explicitly specifies some other number).

show listsize

Display the number of lines that list prints.

Repeating a **list** command with RET discards the argument, so it is equivalent to typing just list. This is more useful than listing the same lines again. An exception is made for an argument of ‘-'; that argument is preserved in repetition so that each repetition moves up in the source file.

In general, the **list** command expects you to supply zero, one or two linespecs. Linespecs specify source lines; there are several ways of writing them, but the effect is always to specify some source line. Here is a complete description of the possible arguments for **list**:

list linespec

Print lines centered around the line specified by linespec.

list first,last

Print lines from first to last. Both arguments are linespecs.

list ,last

Print lines ending with last.

list first,

Print lines starting with first.

list +

Print lines just after the lines last printed.

list -

Print lines just before the lines last printed.

list

As described in the preceding table.

**Here are the ways of specifying a single source line--all the kinds of linespec.**

*number*

Specifies line *number* of the current source file. When a *list* command has two linespecs, this refers to the same source file as the first linespec.

*+offset*

Specifies the line *offset* lines after the last line printed. When used as the second linespec in a **list** command that has two, this specifies the line *offset* lines down from the first linespec.

*-offset*

Specifies the line *offset* lines before the last line printed.

*filename:number*

Specifies line *number* in the source file *filename*.

*function*

Specifies the line that begins the body of the function *function*. For example: in C, this is the line with the open brace.

*filename:function*

Specifies the line of the open-brace that begins the body of the function *function* in the file *filename*. You only need the file name with a function name to avoid ambiguity when there are identically named functions in different source files.

*\*address*

Specifies the line containing the program address *address*. Address may be any expression.

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

On certain operating systems, GDB is able to save a **snapshot** of a program’s state, called a **checkpoint**, and come back to it later.

Returning to a checkpoint effectively undoes everything that has happened in the program since the **checkpoint** was saved. This includes changes in memory, registers, and even (within some limits) system state. Effectively, it is like going back in time to the moment when the checkpoint was saved.

Thus, if you’re stepping through a program and you think you’re getting close to the point where things go wrong, you can save a checkpoint. Then, if you accidentally go too far and miss the critical statement, instead of having to restart your program from the beginning, you can just go back to the checkpoint and start again from there.

This can be especially useful if it takes a lot of time or steps to reach the point where you think the bug occurs.

To use the **checkpoint/restart** method of debugging:

checkpoint

Saves a snapshot of the debugged program’s current execution state. The **checkpoint** command takes no arguments, but each checkpoint is assigned a small integer id, similar to a breakpoint id.

info checkpoints

List the checkpoints that have been saved in the current debugging session. For each checkpoint, the following information will be listed:

Checkpoint ID, Process ID, Code Address, Source line, or label

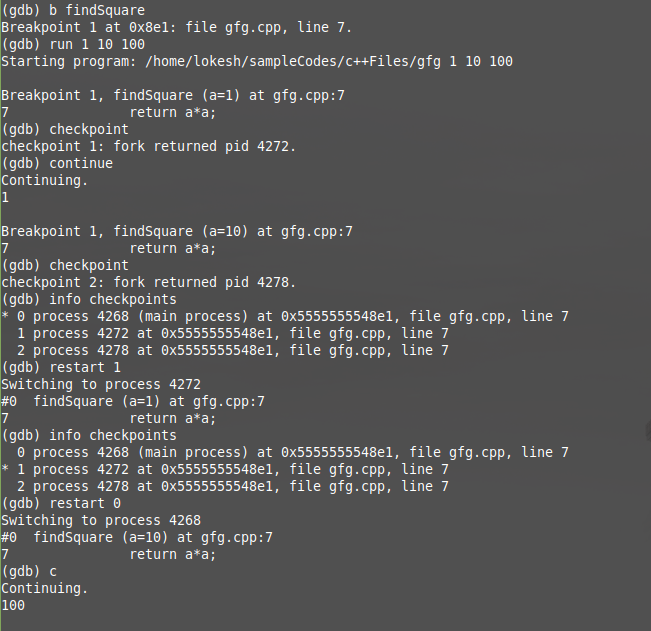
restart <checkpoint-id>

Restore the program state that was saved as checkpoint number checkpoint-id. All program variables, registers, stack frames etc. will be returned to the values that they had when the checkpoint was saved. In essence, gdb will “**wind back the clock**” to the point in time when the checkpoint was saved.

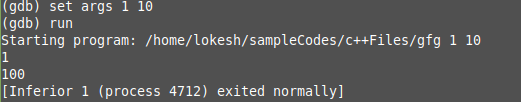
Note that breakpoints, GDB variables, command history etc. are not affected by restoring a checkpoint. In general, a checkpoint only restores things that reside in the program being debugged, not in the debugger.

delete checkpoint <checkpoint-id>

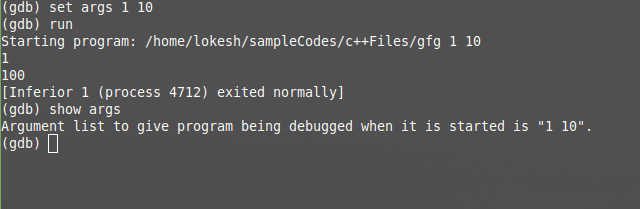
Delete the previously-saved checkpoint identified by checkpoint-id.



**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.



**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.



**display [/format specifier] [expression] and undisplay [display id1] [display id2] …** : These command enables automatic displaying of expressions each time whenever the execution encounters a **breakpoint** or the **n** command. The **undisplay** command is used to remove display expressions. Valid format specifiers are as follows:

o - octal

x - hexadecimal

d - decimal

u - unsigned decimal

t - binary

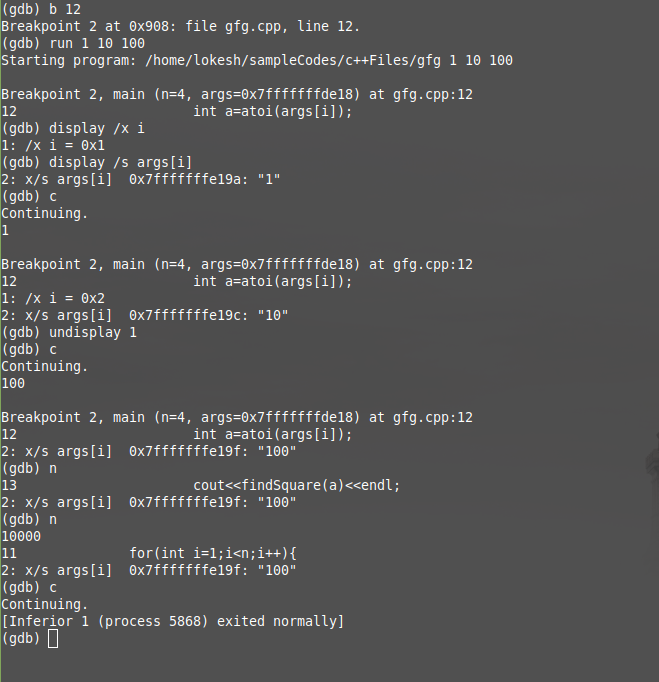
f - floating point

a - address

c - char

s - string

i - instruction



In the above example, the breakpoint is set at line 12 and ran with the arguments 1 10 100. Once the breakpoint is encountered, display command is executed to print the value of i in hexadecimal form and value of args[i] in the string form. After then, whenever the command **n** or a breakpoint is encountered, the values are displayed again until they are disabled using **undisplay** command.

**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.

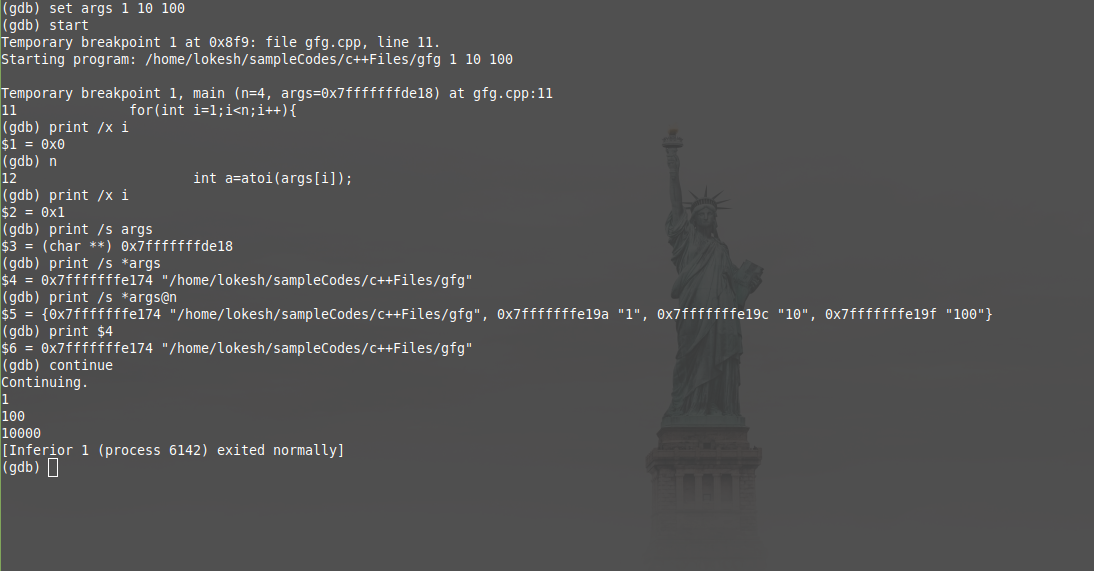
print [Expression]

print $[Previous value number]

print {[Type]}[Address]

print [First element]@[Element count]

print /[Format] [Expression]



**watchpoint** :

Watchpoints are similar to breakpoints. However, watchpoints are not set for functions or lines of code. Watchpoints are set on variables. When those variables are read or written or changes values, the watchpoint is triggered and program execution stops.

#include <stdio.h>

int main(int argc, char \*\*argv)

{

int x = 30;

int y = 10;

x = y;

return 0;

}

**watch** :Use the **watch** command. The argument to the watch command is an expression that is evaluated. This implies that the variabel you want to set a watchpoint on must be in the current scope. So, to set a watchpoint on a non-global variable, you must have set a breakpoint that will stop your program when the variable is in scope. You set the watchpoint after the program breaks.

\*NOTE\* You may notice in the example below that the line of code printed doesn't match with the line that changes the variable x. This is because the store instruction that sets off the watchpoint is the last in the sequence necessary to do the 'x=y' assignment. So the debugger has already gone on to the next line of code. In the examples, a breakpoint has been set on the 'main' function and has been triggered to stop the program.

(gdb) watch x

Hardware watchpoint 4: x

(gdb) c

Continuing.

Hardware watchpoint 4: x

Old value = -1073743192

New value = 11

main (argc=1, argv=0xbffffaf4) at test.c:10

10 return 0;

### **How do I set a read watchpoint for a variable?**

Use the **rwatch** command. Usage is identical to the watch command.

(gdb) rwatch y

Hardware read watchpoint 4: y

(gdb) continue

Continuing.

Hardware read watchpoint 4: y

Value = 1073792976

main (argc=1, argv=0xbffffaf4) at test.c:8

8 x = y;

### **How do I set a read/write watchpoint for a variable?**

Use the **awatch** command. Usage is identical to the watch command.

### **How do I disable watchpoints?**

Active watchpoints show up the breakpoint list. Use the **info breakpoints** command to get this list. Then use the **disable** command to turn off a watchpoint, just like disabling a breakpoint.

(gdb) info breakpoints

Num Type Disp Enb Address What

1 breakpoint keep y 0x080483c6 in main at test.c:5

breakpoint already hit 1 time

4 hw watchpoint keep y x

breakpoint already hit 1 time

(gdb) disable 4