# Running Programs Under GDB

When you run a program under GDB, you must first generate debugging information when you compile it. You may start it with its arguments, if any, in an environment of your choice. You may redirect your program's input and output, debug an already running process, or kill a child process.

## Compiling for debugging

In order to debug a program effectively, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

To request debugging information, **specify the ‘-g’** option when you run the compiler.

Many C compilers are unable to handle the `-g' and `-O' options together. Using those compilers, you cannot generate optimized executables containing debugging information.

When you debug a program compiled with `-g -O', remember that the optimizer is rearranging your code; the debugger shows you what is really there. Do not be too surprised when the execution path does not exactly match your source file!

An extreme example: if you define a variable, but never use it, GDB never sees that variable--because the compiler optimizes it out of existence.

Some things do not work as well with `-g -O' as with just `-g', particularly on machines with instruction scheduling. If in doubt, recompile with `-g' alone, and if this fixes the problem, please report it as a bug (including a test case!).

**-gg**  GDB no longer supports this format.

## Starting your program

### run (r)

Use the run command to start your program under GDB. You must first specify the program name (except on VxWorks) with an argument to GDB (see section Getting In and Out of GDB), or by using the file or exec-file command (see section Commands to specify files).

If you are running your program in an execution environment that supports processes, run creates an inferior process and makes that process run your program. (In environments without processes, run jumps to the start of your program.)

The execution of a program is affected by certain information it receives from its superior. This information may be divided into four categories:

* The arguments

Specify the arguments to give your program as the arguments of the run command.

* The environment

Your program normally inherits its environment from GDB, but you can use the GDB commands set environment and unset environment to change parts of the environment that affect your program.

* The working directory

Your program inherits its working directory from GDB. You can set the GDB working directory with the cd command in GDB.

* The standard input and output.

Your program normally uses the same device for standard input and standard output as GDB is using. You can redirect input and output in the run command line, or you can use the tty command to set a different device for your program. See section Your program's input and output.

Warning: While input and output redirection work, you cannot use pipes to pass the output of the program you are debugging to another program; if you attempt this, GDB is likely to wind up debugging the wrong program.

Once your program has stopped, you may call functions in your program, using the print or call commands. See section Examining Data.

If the modification time of your symbol file has changed since the last time GDB read its symbols, GDB discards its symbol table, and reads it again. When it does this, GDB tries to retain your current breakpoints.

## Your program's environment

The environment consists of a set of environment variables and their values. Environment variables conventionally record such things as your user name, your home directory, your terminal type, and your search path for programs to run. Usually you set up environment variables with the shell and they are inherited by all the other programs you run. When debugging, it can be useful to try running your program with a modified environment without having to start GDB over again.

### path directory

Add directory to the front of the PATH environment variable (the search path for executables), for both GDB and your program. You may specify several directory names, separated by `:' or whitespace. If directory is already in the path, it is moved to the front, so it is searched sooner.

You can use the string `$cwd' to refer to whatever is the current working directory at the time GDB searches the path. If you use `.' instead, it refers to the directory where you executed the path command. GDB replaces `.' in the directory argument (with the current path) before adding directory to the search path.

### show paths

Display the list of search paths for executables (the PATH environment variable).

### show environment [varname]

Print the value of environment variable varname to be given to your program when it starts. If you do not supply varname, print the names and values of all environment variables to be given to your program. **You can abbreviate environment as env.**

### set environment varname [=] value

Set environment variable varname to value. The value changes for your program only, not for GDB itself. value may be any string; the values of environment variables are just strings, and any interpretation is supplied by your program itself. The value parameter is optional; if it is eliminated, the variable is set to a null value.

For example : set env USER = foo

tells a Unix program, when subsequently run, that its user is named `foo'. (The spaces around `=' are used for clarity here; they are not actually required.)

### unset environment varname

Remove variable varname from the environment to be passed to your program. This is different from `set env varname ='; unset environment removes the variable from the environment, rather than assigning it an empty value.

Warning: GDB runs your program using the shell indicated by your SHELL environment variable if it exists (or /bin/sh if not). If your SHELL variable names a shell that runs an initialization file--such as `.cshrc' for C-shell, or `.bashrc' for BASH--any variables you set in that file affect your program. You may wish to move setting of environment variables to files that are only run when you sign on, such as `.login' or `.profile'.

## Your program's arguments

The arguments to your program can be specified by the arguments of the run command. They are passed to a shell, which expands wildcard characters and performs redirection of I/O, and thence to your program. Your SHELL environment variable (if it exists) specifies what shell GDB uses. If you do not define SHELL, GDB uses /bin/sh.

run with no arguments uses the same arguments used by the previous run, or those set by the set args command.

### set args

Specify the arguments to be used the next time your program is run. If set args has no arguments, run executes your program with no arguments. Once you have run your program with arguments, using set args before the next run is the only way to run it again without arguments.

### show args

Show the arguments to give your program when it is started.

## Your program's working directory

Each time you start your program with run, it inherits its working directory from the current working directory of GDB. The GDB working directory is initially whatever it inherited from its parent process (typically the shell).

The GDB working directory also serves as a default for the commands that specify files for GDB to operate on.

### cd directory

Set the GDB working directory to directory.

### pwd

Print the GDB working directory.

## Your program's input and output

By default, the program you run under GDB does input and output to the same terminal that GDB uses. GDB switches the terminal to its own terminal modes to interact with you, but it records the terminal modes your program was using and switches back to them when you continue running your program.

### info terminal

Displays information recorded by GDB about the terminal modes your program is using.

### run > outfile

starts your program, diverting its output to the file `outfile'.

Another way to specify where your program should do input and output is with the tty command. This command accepts a file name as argument, and causes this file to be the default for future run commands. It also resets the controlling terminal for the child process, for future run commands. For example,

### tty /dev/ttyb

directs that processes started with subsequent run commands default to do input and output on the terminal `/dev/ttyb' and have that as their controlling terminal.

An explicit redirection in run overrides the tty command's effect on the input/output device, but not its effect on the controlling terminal.

When you use the tty command or redirect input in the run command, only the input for your program is affected. The input for GDB still comes from your terminal.

## Debugging an already-running process

### attach process-id

This command attaches to a running process--one that was started outside GDB. (info files shows your active targets.) The command takes as argument a process ID. The usual way to find out the process-id of a Unix process is with the ps utility, or with the `jobs -l' shell command.

attach does not repeat if you press RET a second time after executing the command.

To use attach, your program must be running in an environment which supports processes; for example, attach does not work for programs on bare-board targets that lack an operating system. You must also have permission to send the process a signal.

When using attach, you should first use the file command to specify the program running in the process and load its symbol table.

The first thing GDB does after arranging to debug the specified process is to stop it. You can examine and modify an attached process with all the GDB commands that are ordinarily available when you start processes with run. You can insert breakpoints; you can step and continue; you can modify storage. If you would rather the process continue running, you may use the continue command after attaching GDB to the process.

### detach

When you have finished debugging the attached process, you can use the detach command to release it from GDB control. Detaching the process continues its execution. After the detach command, that process and GDB become completely independent once more, and you are ready to attach another process or start one with run. detach does not repeat if you press RET again after executing the command.

If you exit GDB or use the run command while you have an attached process, you kill that process. By default, GDB asks for confirmation if you try to do either of these things; you can control whether or not you need to confirm by using the set confirm command (see section Optional warnings and messages).

## Killing the child process

### kill

Kill the child process in which your program is running under GDB.

This command is useful if you wish to debug a core dump instead of a running process. GDB ignores any core dump file while your program is running.

On some operating systems, a program cannot be executed outside GDB while you have breakpoints set on it inside GDB. You can use the kill command in this situation to permit running your program outside the debugger.

The kill command is also useful if you wish to recompile and relink your program, since on many systems it is impossible to modify an executable file while it is running in a process. In this case, when you next type run, GDB notices that the file has changed, and reads the symbol table again (while trying to preserve your current breakpoint settings).

## Additional process information

Some operating systems provide a facility called `/proc' that can be used to examine the image of a running process using file-system subroutines. If GDB is configured for an operating system with this facility, the command info proc is available to report on several kinds of information about the process running your program.

### info proc

Summarize available information about the process.

### info proc mappings

Report on the address ranges accessible in the program, with information on whether your program may read, write, or execute each range.

### info proc times

Starting time, user CPU time, and system CPU time for your program and its children.

### info proc id

Report on the process IDs related to your program: its own process ID, the ID of its parent, the process group ID, and the session ID.

### info proc status

General information on the state of the process. If the process is stopped, this report includes the reason for stopping, and any signal received.

### info proc all

Show all the above information about the process.

# Stopping and Continuing

Inside GDB, your program may stop for any of several reasons, such as a signal, a breakpoint, or reaching a new line after a GDB command such as step.

### info program

Display information about the status of your program: whether it is running or not, what process it is, and why it stopped.

## Breakpoints, Watchpoints, and Exceptions

A breakpoint makes your program stop whenever a certain point in the program is reached. You can set breakpoints with the break command and its variants (see section Setting breakpoints), to specify the place where your program should stop **by line number, function name or exact address in the program**. In languages with exception handling (such as GNU C++), you can also **set breakpoints where an exception is raised** (see section Breakpoints and exceptions). For each breakpoint, you can add conditions to control in finer detail whether your program stops.

A watchpoint is a special breakpoint that stops your program when the value of an expression changes. You must use a different command to set watchpoints (see section Setting watchpoints), but aside from that, you can manage a watchpoint like any other breakpoint: you enable, disable, and delete both breakpoints and watchpoints using the same commands.

You can arrange to have values from your program displayed automatically whenever GDB stops at a breakpoint. See section Automatic display.

GDB assigns a number to each breakpoint or watchpoint when you create it; these numbers are successive integers starting with one. In many of the commands for controlling various features of breakpoints you use the breakpoint number to say which breakpoint you want to change. Each breakpoint may be enabled or disabled; if disabled, it has no effect on your program until you enable it again.

## Setting breakpoints

Breakpoints are set with the break command (abbreviated b). The debugger convenience variable `$bpnum' records the number of the beakpoint you've set most recently; see section Convenience variables, for a discussion of what you can do with convenience variables.

You have several ways to say where the breakpoint should go.

### break function

Set a breakpoint at entry to function function. When using source languages that permit overloading of symbols, such as C++, function may refer to more than one possible place to break. See section Breakpoint menus, for a discussion of that situation.

### break +offset

### break -offset

Set a breakpoint some number of lines forward or back from the position at which execution stopped in the currently selected frame.

### break linenum

Set a breakpoint at line linenum in the current source file. That file is the last file whose source text was printed. This breakpoint stops your program just before it executes any of the code on that line.

### break filename:linenum

Set a breakpoint at line linenum in source file filename.

### break filename:function

Set a breakpoint at entry to function function found in file filename. Specifying a file name as well as a function name is superfluous except when multiple files contain similarly named functions.

### break \*address

Set a breakpoint at address address. You can use this to set breakpoints in parts of your program which do not have debugging information or source files.

### break

When called without any arguments, break sets a breakpoint at the next instruction to be executed in the selected stack frame (see section Examining the Stack). In any selected frame but the innermost, this makes your program stop as soon as control returns to that frame. This is similar to the effect of a finish command in the frame inside the selected frame--except that finish does not leave an active breakpoint. If you use break without an argument in the innermost frame, GDB stops the next time it reaches the current location; this may be useful inside loops.

GDB normally ignores breakpoints when it resumes execution, until at least one instruction has been executed. If it did not do this, you would be unable to proceed past a breakpoint without first disabling the breakpoint. This rule applies whether or not the breakpoint already existed when your program stopped.

### break ... if cond

Set a breakpoint with condition cond; evaluate the expression cond each time the breakpoint is reached, and stop only if the value is nonzero--that is, if cond evaluates as true. `...' stands for one of the possible arguments described above (or no argument) specifying where to break. See section Break conditions, for more information on breakpoint conditions.

### tbreak args

Set a breakpoint enabled only for one stop. args are the same as for the break command, and the breakpoint is set in the same way, but the breakpoint is automatically deleted after the first time your program stops there. See section Disabling breakpoints.

### rbreak regex

Set breakpoints on all functions matching the regular expression regex. This command sets an unconditional breakpoint on all matches, printing a list of all breakpoints it set. Once these breakpoints are set, they are treated just like the breakpoints set with the break command. You can delete them, disable them, or make them conditional the same way as any other breakpoint.

When debugging C++ programs, rbreak is useful for setting breakpoints on overloaded functions that are not members of any special classes.

### info breakpoints [n]

### info break [n]

### info watchpoints [n]

Print a table of all breakpoints and watchpoints set and not deleted, with the following columns for each breakpoint:

|  |  |
| --- | --- |
| Breakpoint Numbers  Type | Breakpoint or watchpoint. |
| Disposition | Whether the breakpoint is marked to be disabled or deleted when hit. |
| Enabled or Disabled | Enabled breakpoints are marked with `y'. `n' marks breakpoints that are not enabled. |
| Address | Where the breakpoint is in your program, as a memory address. |
| What | Where the breakpoint is in the source for your program, as a file and line number. |

If a breakpoint is conditional, info break shows the condition on the line following the affected breakpoint; breakpoint commands, if any, are listed after that.

info break with a breakpoint number n as argument lists only that breakpoint. The convenience variable $\_ and the default examining-address for the x command are set to the address of the last breakpoint listed (see section Examining memory).

GDB allows you to set any number of breakpoints at the same place in your program. There is nothing silly or meaningless about this. When the breakpoints are conditional, this is even useful (see section Break conditions).

GDB itself sometimes sets breakpoints in your program for special purposes, such as proper handling of longjmp (in C programs). These internal breakpoints are assigned negative numbers, starting with -1; `info breakpoints' does not display them.

### maint info breakpoints

Using the same format as `info breakpoints', display both the breakpoints you've set explicitly, and those GDB is using for internal purposes. Internal breakpoints are shown with negative breakpoint numbers. The type column identifies what kind of breakpoint is shown:

|  |  |
| --- | --- |
| Breakpoint | Normal, explicitly set breakpoint. |
| watchpoint | Normal, explicitly set watchpoint. |
| longjmp | Internal breakpoint, used to handle correctly stepping through longjmp calls. |
| longjmp resume | Internal breakpoint at the target of a longjmp. |
| until | Temporary internal breakpoint used by the GDB until command. |
| finish | Temporary internal breakpoint used by the GDB finish command. |

## Setting watchpoints

You can use a watchpoint to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen.

Some processors provide special hardware to support watchpoint evaluation; GDB will use such hardware if it is available, and if the support code has been added for that configuration.

### watch expr

Set a watchpoint for an expression.

### info watchpoints

This command prints a list of watchpoints and breakpoints; it is the same as info break.

Warning: in multi-thread programs, watchpoints have only limited usefulness. With the current watchpoint implementation, GDB can only watch the value of an expression in a single thread. If you are confident that the expression can only change due to the current thread's activity (and if you are also confident that no other thread can become current), then you can use watchpoints as usual. However, GDB may not notice when a non-current thread's activity changes the expression.

## Breakpoints and exceptions

You can use GDB to examine what caused your program to raise an exception, and to list the exceptions your program is prepared to handle at a given point in time.

### catch exceptions

You can set breakpoints at active exception handlers by using the catch command. exceptions is a list of names of exceptions to catch.

### info catch

You can use info catch to list active exception handlers. See section Information about a frame.

There are currently some limitations to exception handling in GDB:

* If you call a function interactively, GDB normally returns control to you when the function has finished executing. If the call raises an exception, however, the call may bypass the mechanism that returns control to you and cause your program to simply continue running until it hits a breakpoint, catches a signal that GDB is listening for, or exits.
* You cannot raise an exception interactively.
* You cannot install an exception handler interactively.

Sometimes catch is not the best way to debug exception handling: if you need to know exactly where an exception is raised, it is better to stop before the exception handler is called, since that way you can see the stack before any unwinding takes place. If you set a breakpoint in an exception handler instead, it may not be easy to find out where the exception was raised.

To stop just before an exception handler is called, you need some knowledge of the implementation. In the case of GNU C++, exceptions are raised by calling a library function named **\_\_raise\_exception** which has the following ANSI C interface:

/\* addr is where the exception identifier is stored.

ID is the exception identifier. \*/

**void \_\_raise\_exception (void \*\*addr, void \*id);**

To make the debugger catch all exceptions before any stack unwinding takes place, set a breakpoint on \_\_raise\_exception (see section Breakpoints, watchpoints, and exceptions).

With a conditional breakpoint (see section Break conditions) that depends on the value of id, you can stop your program when a specific exception is raised. You can use multiple conditional breakpoints to stop your program when any of a number of exceptions are raised.

## Deleting breakpoints

clear command: delete breakpoints according to where they are in your program.

delete command: delete individual breakpoints or watchpoints by specifying their breakpoint numbers.

GDB automatically ignores breakpoints on the first instruction to be executed when you continue execution without changing the execution address.

### clear

Delete any breakpoints at the next instruction to be executed in the selected stack frame (see section Selecting a frame). When the innermost frame is selected, this is a good way to delete a breakpoint where your program just stopped.

### clear function

### clear filename:function

Delete any breakpoints set at entry to the function function.

### clear linenum

### clear filename:linenum

Delete any breakpoints set at or within the code of the specified line.

### delete [breakpoints] [bnums...]

### d

Delete the breakpoints or watchpoints of the numbers specified as arguments. If no argument is specified, delete all breakpoints (GDB asks confirmation, unless you have set confirm off). You can abbreviate this command as d.

## Disabling breakpoints

Rather than deleting a breakpoint or watchpoint, you might prefer to disable it. This makes the breakpoint inoperative as if it had been deleted, but remembers the information on the breakpoint so that you can enable it again later.

You disable and enable breakpoints and watchpoints with the enable and disable commands, optionally specifying one or more breakpoint numbers as arguments. Use info break or info watch to print a list of breakpoints or watchpoints if you do not know which numbers to use.

A breakpoint or watchpoint can have any of four different states of enablement:

* **Enabled**: The breakpoint stops your program. A breakpoint set with the break command starts out in this state.
* **Disabled**: The breakpoint has no effect on your program.
* **Enabled once**: The breakpoint stops your program, but then becomes disabled. A breakpoint set with the tbreak command starts out in this state.
* **Enabled for deletion**: The breakpoint stops your program, but immediately after it does so it is deleted permanently.

You can use the following commands to enable or disable breakpoints and watchpoints:

### disable [breakpoints] [bnums...]

Disable the specified breakpoints--or all breakpoints, if none are listed. A disabled breakpoint has no effect but is not forgotten. All options such as ignore-counts, conditions and commands are remembered in case the breakpoint is enabled again later. You may abbreviate disable as dis.

### enable [breakpoints] [bnums...]

Enable the specified breakpoints (or all defined breakpoints). They become effective once again in stopping your program.

### enable [breakpoints] once bnums...

Enable the specified breakpoints temporarily. GDB disables any of these breakpoints immediately after stopping your program.

### enable [breakpoints] delete bnums...

Enable the specified breakpoints to work once, then die. GDB deletes any of these breakpoints as soon as your program stops there.

Save for a breakpoint set with tbreak (see section Setting breakpoints).

(The command until can set and delete a breakpoint of its own, but it does not change the state of your other breakpoints; see section Continuing and stepping.)

## Break conditions

The simplest sort of breakpoint breaks every time your program reaches a specified place. A breakpoint with a condition evaluates the expression each time your program reaches it, and your program stops only if the condition is true.

Conditions are also accepted for watchpoints; you may not need them, since a watchpoint is inspecting the value of an expression anyhow--but it might be simpler, say, to just set a watchpoint on a variable name, and specify a condition that tests whether the new value is an interesting one.

Break conditions can have side effects, and may even call functions in your program. Note that breakpoint commands are usually more convenient and flexible for the purpose of performing side effects when a breakpoint is reached (see section Breakpoint command lists).

### condition bnum expression

Specify expression as the break condition for breakpoint or watchpoint number bnum. After you set a condition, breakpoint bnum stops your program only if the value of expression is true (nonzero, in C). When you use condition, GDB checks expression immediately for syntactic correctness, and to determine whether symbols in it have referents in the context of your breakpoint. GDB does not actually evaluate expression at the time the condition command is given.

### condition bnum

Remove the condition from breakpoint number bnum. It becomes an ordinary unconditional breakpoint.

A special case of a breakpoint condition is to stop only when the breakpoint has been reached a certain number of times. If your program reaches a breakpoint whose ignore count is positive, then instead of stopping, it just decrements the ignore count by one and continues. As a result, if the ignore count value is n, the breakpoint does not stop the next n times your program reaches it.

### ignore bnum count

Set the ignore count of breakpoint number bnum to count. The next count times the breakpoint is reached, your program's execution does not stop; other than to decrement the ignore count, GDB takes no action.

To make the breakpoint stop the next time it is reached, specify a count of zero.

When you use continue to resume execution of your program from a breakpoint, you can specify an ignore count directly as an argument to continue, rather than using ignore. See section Continuing and stepping.

If a breakpoint has a positive ignore count and a condition, the condition is not checked. Once the ignore count reaches zero, GDB resumes checking the condition.

You could achieve the effect of the ignore count with a condition such as `$foo-- <= 0' using a debugger convenience variable that is decremented each time. See section Convenience variables.

## Breakpoint command lists

You can give any breakpoint (or watchpoint) a series of commands to execute when your program stops due to that breakpoint. For example, you might want to print the values of certain expressions, or enable other breakpoints.

### commands [bnum]

### ... command-list ...

### end

Specify a list of commands for breakpoint number bnum. The commands themselves appear on the following lines. Type a line containing just end to terminate the commands.

To remove all commands from a breakpoint, type commands and follow it immediately with end; that is, give no commands.

With no bnum argument, commands refers to the last breakpoint or watchpoint set (not to the breakpoint most recently encountered).

Pressing RET as a means of repeating the last GDB command is disabled within a command-list.

You can use breakpoint commands to start your program up again. Simply use the continue command, or step, or any other command that resumes execution.

Any other commands in the command list, after a command that resumes execution, are ignored. This is because any time you resume execution (even with a simple next or step), you may encounter another breakpoint--which could have its own command list, leading to ambiguities about which list to execute.

If the first command you specify in a command list is silent. This may be desirable for breakpoints that are to print a specific message and then continue. silent is meaningful only at the beginning of a breakpoint command list.

The commands echo, output, and printf allow you to print precisely controlled output, and are often useful in silent breakpoints. See section Commands for controlled output.

For example, here is how you could use breakpoint commands to print the value of x at entry to foo whenever x is positive.

*break foo if x>0*

*commands*

*silent*

*printf "x is %d\n",x*

*cont*

*end*

One application for breakpoint commands is to compensate for one bug so you can test for another. Put a breakpoint just after the erroneous line of code, give it a condition to detect the case in which something erroneous has been done, and give it commands to assign correct values to any variables that need them. End with the continue command so that your program does not stop, and start with the silent command so that no output is produced. Here is an example:

*break 403*

*commands*

*silent*

*set x = y + 4*

*cont*

*end*

## Breakpoint menus

When a function name is overloaded, `break function' is not enough to tell GDB where you want a breakpoint. If you realize this is a problem, you can use something like `break function(types)' to specify which particular version of the function you want. Otherwise, GDB offers you a menu of numbered choices for different possible breakpoints, and waits for your selection with the prompt `>'. The first two options are always `[0] cancel' and `[1] all'. Typing 1 sets a breakpoint at each definition of function, and typing 0 aborts the break command without setting any new breakpoints.

For example, the following session excerpt shows an attempt to set a breakpoint at the overloaded symbol String::after. We choose three particular definitions of that function name:

*(gdb) b String::after*

*[0] cancel*

*[1] all*

*[2] file:String.cc; line number:867*

*[3] file:String.cc; line number:860*

*[4] file:String.cc; line number:875*

*[5] file:String.cc; line number:853*

*[6] file:String.cc; line number:846*

*[7] file:String.cc; line number:735*

*> 2 4 6*

*Breakpoint 1 at 0xb26c: file String.cc, line 867.*

*Breakpoint 2 at 0xb344: file String.cc, line 875.*

*Breakpoint 3 at 0xafcc: file String.cc, line 846.*

*Multiple breakpoints were set.*

*Use the "delete" command to delete unwanted*

*breakpoints.*

*(gdb)*

## "Cannot insert breakpoints"

Under some operating systems, breakpoints cannot be used in a program if any other process is running that program. In this situation, attempting to run or continue a program with a breakpoint causes GDB to stop the other process.

When this happens, you have three ways to proceed:

1. Remove or disable the breakpoints, then continue.
2. Suspend GDB, and copy the file containing your program to a new name. Resume GDB and use the exec-file command to specify that GDB should run your program under that name. Then start your program again.
3. Relink your program so that the text segment is nonsharable, using the linker option `-N'. The operating system limitation may not apply to nonsharable executables.

## Continuing and stepping

Continuing means resuming program execution until your program completes normally. In contrast, stepping means executing just one more "step" of your program, where "step" may mean either one line of source code, or one machine instruction (depending on what particular command you use). Either when continuing or when stepping, your program may stop even sooner, due to a breakpoint or a signal. (If due to a signal, you may want to use handle, or use `signal 0' to resume execution. See section Signals.)

### continue [ignore-count]

### c [ignore-count]

### fg [ignore-count]

Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument ignore-count allows you to specify a further number of times to ignore a breakpoint at this location; its effect is like that of ignore (see section Break conditions).

The argument ignore-count is meaningful only when your program stopped due to a breakpoint. At other times, the argument to continue is ignored.

The synonyms c and fg are provided purely for convenience, and have exactly the same behavior as continue.

To resume execution at a different place, you can use return (see section Returning from a function) to go back to the calling function; or jump (see section Continuing at a different address) to go to an arbitrary location in your program.

A typical technique for using stepping is to set a breakpoint.

### step

### s

Continue running your program until control reaches a different source line, then stop it and return control to GDB. This command is abbreviated s.

Warning: If you use the step command while control is within a function that was compiled without debugging information, execution proceeds until control reaches a function that does have debugging information. Likewise, it will not step into a function which is compiled without debugging information. To step through functions without debugging information, use the stepi command, described below.

### step count

Continue running as in step, but do so count times. If a breakpoint is reached, or a signal not related to stepping occurs before count steps, stepping stops right away.

### next [count]

Continue to the next source line in the current (innermost) stack frame. Similar to step, but any function calls appearing within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the stack level which was executing when the next command was given. This command is abbreviated n.

An argument count is a repeat count, as for step.

next within a function that lacks debugging information acts like step, but any function calls appearing within the code of the function are executed without stopping.

### finish

Continue running until just after function in the selected stack frame returns. Print the returned value (if any).

Contrast this with the return command (see section Returning from a function).

### until

### u

Continue running until a source line past the current line, in the current stack frame, is reached. This command is used to avoid single stepping through a loop more than once. It is like the next command, except that when until encounters a jump, it automatically continues execution until the program counter is greater than the address of the jump.

This means that when you reach the end of a loop after single stepping though it, until makes your program continue execution until it exits the loop. In contrast, a next command at the end of a loop simply steps back to the beginning of the loop, which forces you to step through the next iteration.

until always stops your program if it attempts to exit the current stack frame.

until may produce somewhat counterintuitive results if the order of machine code does not match the order of the source lines. For example, in the following excerpt from a debugging session, the f (frame) command shows that execution is stopped at line 206; yet when we use until, we get to line 195:

*(gdb) f*

*#0 main (argc=4, argv=0xf7fffae8) at m4.c:206*

*206 expand\_input();*

*(gdb) until*

*195 for ( ; argc > 0; NEXTARG) {*

This happened because, for execution efficiency, the compiler had generated code for the loop closure test at the end, rather than the start, of the loop--even though the test in a C for-loop is written before the body of the loop. The until command appeared to step back to the beginning of the loop when it advanced to this expression; however, it has not really gone to an earlier statement--not in terms of the actual machine code.

until with no argument works by means of single instruction stepping, and hence is slower than until with an argument.

### until location

### u location

Continue running your program until either the specified location is reached, or the current stack frame returns. location is any of the forms of argument acceptable to break (see section Setting breakpoints). This form of the command uses breakpoints, and hence is quicker than until without an argument.

### stepi [count]

### si [count]

Execute one machine instruction, then stop and return to the debugger.

It is often useful to do `display/i $pc' when stepping by machine instructions. This makes GDB automatically display the next instruction to be executed, each time your program stops. See section Automatic display.

An argument is a repeat count, as in step.

### nexti [count]

### ni [count]

Execute one machine instruction, but if it is a function call, proceed until the function returns.

An argument is a repeat count, as in next.