# 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.

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

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?

## 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.

## Note

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.

# 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.

Symbol tables may be embedded into the program or stored as a separate file. So if you plan to debug your program, then it is required to create a symbol table which will have the required information to debug the program.

We can infer the following facts about symbol tables:

* A symbol table works for a particular version of the program – if the program changes, a new table must be created.
* Debug builds are often larger and slower than retail (non-debug) builds; debug builds contain the symbol table and other ancillary information.
* If you wish to debug a binary program you did not compile yourself, you must get the symbol tables from the author.

To let GDB be able to read all that information line by line from the symbol table, we need to compile it a bit differently. Normally we compile our programs as:

gcc hello.cc -o hello

Instead of doing this, we need to compile with the -g flag as shown below:

gcc -g hello.cc -o hello

# Getting In and Out of GDB

How to start GDB, and how to get out of it ?

## Invoking GDB

Invoke GDB by running the program gdb.

Once started, GDB reads commands from the terminal until you tell it to exit.

The most usual way to start GDB is with one argument, specifying an executable program:

### gdb program

You can also start with both an executable program and a core file specified:

### gdb program core

You can, instead, specify a process ID as a second argument, if you want to debug a running process:

### gdb program 1234

would attach GDB to process 1234 (unless you also have a file named `1234'; GDB does check for a core file first).

### gdb -help (gdb -h)

to display all available options and briefly describe their use

All options and command line arguments you give are processed in sequential order. The order makes a difference when the `-x' option is used.

When GDB starts, it reads any arguments other than options as specifying an executable file and core file (or process ID). This is the same as if the arguments were specified by the `-se' and `-c' options respectively. (GDB reads the first argument that does not have an associated option flag as equivalent to the `-se' option followed by that argument; and the second argument that does not have an associated option flag, if any, as equivalent to the `-c' option followed by that argument.)

Many options have both long and short forms. GDB also recognizes the long forms if you truncate them, so long as enough of the option is present to be unambiguous. (If you prefer, you can flag option arguments with `--' rather than `-', though we illustrate the more usual convention.)

## Choosing files

### -symbols file (-s file)

Read symbol table from file file.

### -exec file (-e file)

Use file file as the executable file to execute when appropriate, and for examining pure data in conjunction with a core dump.

### -se file

Read symbol table from file file and use it as the executable file.

### -core file (-c file)

Use file file as a core dump to examine.

### -c number

Connect to process ID number, as with the attach command

(unless there is a file in core-dump format named number, in which case `-c' specifies that file as a core dump to read).

### -command file (-x file)

Execute GDB commands from file file. See section Command files.

The order makes a difference when the `-x' option is used.

### -directory directory (-d directory)

Add directory to the path to search for source files.

### -mapped (-m)

Warning: this option depends on operating system facilities that are not supported on all systems.

If memory-mapped files are available on your system through the mmap system call, you can use this option to have GDB write the symbols from your program into a reusable file in the current directory. If the program you are debugging is called `/tmp/fred', the mapped symbol file is `./fred.syms'. Future GDB debugging sessions notice the presence of this file, and can quickly map in symbol information from it, rather than reading the symbol table from the executable program.

The `.syms' file is specific to the host machine where GDB is run. It holds an exact image of the internal GDB symbol table. It cannot be shared across multiple host platforms.

### -readnow (-r)

Read each symbol file's entire symbol table immediately, rather than the default, which is to read it incrementally as it is needed. This makes startup slower, but makes future operations faster.

The -mapped and -readnow options are typically combined in order to build a `.syms' file that contains complete symbol information. (See section Commands to specify files, for information on `.syms' files.) A simple GDB invocation to do nothing but build a `.syms' file for future use is:

### gdb -batch -nx -mapped -readnowprogramname

## Choosing modes

You can run GDB in various alternative modes--for example, in batch mode or quiet mode.

### -nx (-n)

Do not execute commands from any initialization files (normally called `.gdbinit'). Normally, the commands in these files are executed after all the command options and arguments have been processed. See section Command files.

### -quiet (-q)

"Quiet". Do not print the introductory and copyright messages. These messages are also suppressed in batch mode.

### -batch

Run in batch mode.

Exit with status 0 after processing all the command files specified with `-x' (and all commands from initialization files, if not inhibited with `-n').

Exit with nonzero status if an error occurs in executing the GDB commands in the command files.

Batch mode may be useful for running GDB as a filter, for example to download and run a program on another computer, in order to make this more useful, the message

Program exited normally

is not issued when running in batch mode.

(which is ordinarily issued whenever a program running under GDB control terminates)

### -cd directory

Run GDB using directory as its working directory, instead of the current directory.

### -fullname (-f)

Emacs sets this option when it runs GDB as a subprocess. It tells GDB to output the full file name and line number in a standard, recognizable fashion each time a stack frame is displayed (which includes each time your program stops).

This recognizable format looks like two `\032' characters, followed by the file name, line number and character position separated by colons, and a newline. The Emacs-to-GDB interface program uses the two `\032' characters as a signal to display the source code for the frame.

### -b bps

Set the line speed (baud rate or bits per second) of any serial interface used by GDB for remote debugging.

### -tty device

Run using device for your program's standard input and output.

## Quitting GDB

### quit (q)

### C-d

To exit GDB, use the quit command (abbreviated q), or type an end-of-file character (usually C-d).

An interrupt (often C-c) does not exit from GDB, but rather terminates the action of any GDB command that is in progress and returns to GDB command level. It is safe to type the interrupt character at any time because GDB does not allow it to take effect until a time when it is safe.

If you have been using GDB to control an attached process or device, you can release it with the detach command (see section Debugging an already-running process).

## Shell commands

If you need to execute occasional shell commands during your debugging session, there is no need to leave or suspend GDB; you can just use the shell command.

### shell command string

Invoke a the standard shell to execute command string. If it exists, the environment variable SHELL determines which shell to run. Otherwise GDB uses /bin/sh.

The utility make is often needed in development environments. You do not have to use the shell command for this purpose in GDB:

### make make-args

Execute the make program with the specified arguments.

This is equivalent to `shell make make-args'.

# GDB Commands

You can abbreviate a GDB command to the first few letters of the command name, if that abbreviation is unambiguous.

In some cases, even ambiguous abbreviations are allowed.

You can repeat certain GDB commands by typing just RET.

You can also use the TAB key to get GDB to fill out the rest of a word in a command (or to show you the alternatives available, if there is more than one possibility).

## Command syntax

A GDB command is a single line of input. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. Some command names do not allow any arguments.

For example, the command step accepts an argument which is the number of times to step, as in `step 5'. You can also use the step command with no arguments.

Possible command abbreviations are listed in the documentation for individual commands.

For example, s is specially defined as equivalent to step even though there are other commands whose names start with s.

You can test abbreviations by using them as arguments to the help command.

A blank line as input to GDB (typing just RET) means to repeat the previous command. Certain commands (for example, run) will not repeat this way; these are commands whose unintentional repetition might cause trouble and which you are unlikely to want to repeat.

The list and x commands, when you repeat them with RET, construct new arguments rather than repeating exactly as typed. This permits easy scanning of source or memory.

GDB can also use RET in another way: to partition lengthy output, in a way similar to the common utility more (see section Screen size). Since it is easy to press one RET too many in this situation, GDB disables command repetition after any command that generates this sort of display.

Any text from a # to the end of the line is a comment; it does nothing. This is useful mainly in command files (see section Command files).

## Command completion

GDB can fill in the rest of a word in a command for you, if there is only one possibility; it can also show you what the valid possibilities are for the next word in a command, at any time. This works for GDB commands, GDB subcommands, and the names of symbols in your program.

Press the TAB key whenever you want GDB to fill out the rest of a word. If there is only one possibility, GDB fills in the word, and waits for you to finish the command (or press RET to enter it). For example, if you type

### (gdb) info bre TAB

GDB fills in the rest of the word `breakpoints', since that is the only info subcommand beginning with `bre':

### (gdb) info breakpoints

You can either press RET at this point, to run the info breakpoints command, or backspace and enter something else, if `breakpoints' does not look like the command you expected. (If you were sure you wanted info breakpoints in the first place, you might as well just type RET immediately after `info bre', to exploit command abbreviations rather than command completion).

If there is more than one possibility for the next word when you press TAB, GDB sounds a bell. You can either supply more characters and try again, or just press TAB a second time; GDB displays all the possible completions for that word. For example,

### (gdb) b make\_ TAB

GDB sounds bell; press TAB again, to see:

make\_a\_section\_from\_file make\_environ

make\_abs\_section make\_function\_type

make\_blockvector make\_pointer\_type

make\_cleanup make\_reference\_type

make\_command make\_symbol\_completion\_list

(gdb) b make\_

After displaying the available possibilities, GDB copies your partial input (`b make\_' in the example) so you can finish the command.

If you just want to see the list of alternatives in the first place, you can press M-? rather than pressing TAB twice.

M-? means META ?. You can type this either by holding down a key designated as the META shift on your keyboard (if there is one) while typing ?, or as ESC followed by ?.

Sometimes the string you need, while logically a "word", may contain parentheses or other characters that GDB normally excludes from its notion of a word. To permit word completion to work in this situation, you may enclose words in ' (single quote marks) in GDB commands.

The most likely situation where you might need this is in typing the name of a C++ function. This is because C++ allows function overloading (multiple definitions of the same function, distinguished by argument type). For example,

### (gdb) b 'bubble( M-?

bubble(double,double) bubble(int,int)

### (gdb) b 'bubble(

In some cases, GDB can tell that completing a name requires using quotes. When this happens, GDB inserts the quote for you (while completing as much as it can) if you do not type the quote in the first place:

### (gdb) b bub TAB

GDB alters your input line to the following, and rings a bell:

### (gdb) b 'bubble(

In general, GDB can tell that a quote is needed (and inserts it) if you have not yet started typing the argument list when you ask for completion on an overloaded symbol.

## Getting help

### help (h)

You can use help (abbreviated h) with no arguments to display a short list of named classes of commands:

*(gdb) help*

*List of classes of commands:*

*running -- Running the program*

*stack -- Examining the stack*

*data -- Examining data*

*breakpoints -- Making program stop at certain points*

*files -- Specifying and examining files*

*status -- Status inquiries*

*support -- Support facilities*

*user-defined -- User-defined commands*

*aliases -- Aliases of other commands*

*obscure -- Obscure features*

*Type "help" followed by a class name for a list of*

*commands in that class.*

*Type "help" followed by command name for full*

*documentation.*

*Command name abbreviations are allowed if unambiguous.*

*(gdb)*

### help class

Using one of the general help classes as an argument, you can get a list of the individual commands in that class. For example, here is the help display for the class status:

*(gdb) help status*

*Status inquiries.*

*List of commands:*

*show -- Generic command for showing things set*

*with "set"*

*info -- Generic command for printing status*

*Type "help" followed by command name for full*

*documentation.*

*Command name abbreviations are allowed if unambiguous.*

*(gdb)*

### help command

With a command name as help argument, GDB displays a short paragraph on how to use that command.

In addition to help, you can use the GDB commands info and show to inquire about the state of your program, or the state of GDB itself.

### Info (i)

This command (abbreviated i) is for describing the state of your program.

For example, you can list the arguments given to your program with info args, list the registers currently in use with info registers, or list the breakpoints you have set with info breakpoints. You can get a complete list of the info sub-commands with help info.

### show

In contrast, show is for describing the state of GDB itself. You can change most of the things you can show, by using the related command set; for example, you can control what number system is used for displays with set radix, or simply inquire which is currently in use with show radix.

To display all the settable parameters and their current values, you can use show with no arguments; you may also use info set. Both commands produce the same display.

### show ~ info set

Here are three miscellaneous show subcommands, all of which are exceptional in lacking corresponding set commands:

### show version

Show what version of GDB is running. You should include this information in GDB bug-reports. If multiple versions of GDB are in use at your site. The version number is also announced when you start GDB.

### show copying

Display information about permission for copying GDB.

### show warranty

Display the GNU "NO WARRANTY" statement.