1 The GNU Source-Level Debugger

This chapter describes how to debug a C program using the GNU debugger from the Free Software Foundation (the GNU debugger has been extended in OPENSTEP to support the use of Objective-C).

This chapter provides an overview of the GDB debugger and how to use it. The chapter ends with a discussion of OPENSTEP-specific extensions to GDB. These OPENSTEP extensions provide full compatibility with standard GDB, while offering the following additional features useful for developing programs within the OPENSTEP software environment:

- Additional debugger commands
- Extensions to existing debugger commands
- Support for debugging Objective-C code

This chapter is a modified version of documentation provided by the Free Software Foundation; see the section "Legal Considerations" at the end of the chapter for important related information.

This chapter Copyright © 1988, 1989, 1990, 1991, 1992, 1993, 1994, and 1995 by Free Software Foundation, Inc. and Copyright © 1990, 1991, 1992, 1993, 1994, 1995, 1996, and 1997 by Apple Software, Inc.

Summary of GDB

The purpose of a debugger such as GDB is to allow you to execute another program while examining what's going on inside it. We call the other program "your program" or "the program being debugged."

GDB can do four kinds of things (plus other things in support of these):

- Start the program, specifying anything that might affect its behavior.
- Make the program stop on specified conditions.
- Examine what has happened—when the program has stopped—so you can see bugs happen.
- Change things in the program, so you can correct the effects of one bug and go on to learn about another without having to recompile first.

Compiling Your Program for Debugging

To debug a program effectively, you need to ask for debugging information when you compile it. This information in the object file 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. We recommend that you always use **-g** when you compile a program. You may think the program is correct, but there's no sense in pushing your luck.

The GNU C compiler supports debugging with optimization (by using the **-O** compiler option). Although GDB provides the capability to debug programs compiled with optimization, the debugger may provide confusing or misleading information when debugging optimized programs. The intention is to provide some recourse in those situations where debugging optimized programs is necessary. However, debugging optimized programs should not be done routinely on some processors.

With these warnings in mind, it can still be useful to debug optimized programs, provided that you're aware of the limitations of the debugger in these circumstances. Most importantly, the debugger should be able to provide correct backtraces of your program's function call stack. This is often all that is needed to find the problem. Printing the values of variables, however, may give incorrect results, since the debugger has insufficient information to be sure where a variable resides at any given time. Variables declared **volatile** will always have correct values, and global variables will almost always be correct; local variables, however, are likely to be incorrectly reported.

Variables declared **register** are optimized by the compiler even when optimizing is not requested with the **-O** compiler option—these may also give misleading results. To ensure a completely predictable debugging environment, it's best to compile without the **-O** flag and with the compiler option "**-Dregister=**". This option causes the C preprocessor to effectively delete all **register** declarations from your program for this compilation. (In fact, with the GNU C compiler, there's no need to declare any variables to be **register** variables. When optimizing, the GNU C compiler may place any variable in a register whether it's declared **register** or not. On the other hand, declaring variables to be **register** variables may make it more difficult to debug your program when not optimizing. Therefore, the use of the **register** declaration is discouraged.)

Running GDB

In the OPENSTEP development environment, you're likely to use GDB by running it in the Project Builder Launch panel. In this panel, you enter commands at the GDB prompt, and debugger output appears on subsequent lines. (You can also run GDB as a subprocess in the GNU Emacs editor, as described later in this chapter.) Although Project Builder provides an interface and shortcuts to many common GDB commands, this chapter describes only the GDB command-line interface. For more information on Project Builder's interface for the debugger, see the book *OPENSTEP Development: Tools and Techniques*.

To start GDB from within a shell window, enter the following command:

gdb *name* [*core* | *processID*]

name is the name of your executable program. *core*, if specified, is the name of the core dump file to be examined. *processID* is the ID of an already running process that you want to debug. See the rest of this section for information about optional command-line arguments and switches. Once started, GDB reads commands from the terminal until you quit by giving the **quit** command.

A GDB command is a single line of input. There's no limit to how long it can be. It starts with a command name, optionally followed by arguments (some commands don't allow arguments).

GDB command names may always be abbreviated if the abbreviation is unambiguous. Sometimes even ambiguous abbreviations are allowed. For example, **s** is equivalent to **step** even though there are other commands whose names start with **s**. Possible command abbreviations are stated in the documentation of the individual commands.

A blank line as input to GDB means to repeat the previous command verbatim. Certain commands don't allow themselves to be repeated this way; these are commands for which unintentional repetition might cause trouble and which you're unlikely to want to repeat. Certain others (**list** and **x**) act differently when repeated because that's more useful.

A line of input starting with **#** is a comment; it does nothing. This is useful mainly in command files (see the section "Command Files").

GDB prompts for commands by displaying the **(gdb)** prompt. You can change the prompt with the **set prompt** command (this is most useful when debugging GDB itself):

set prompt *newprompt*

To exit GDB, use the **quit** command (abbreviated **q**) or type Control-D. Control-C won't exit from GDB, but rather will terminate the action of any GDB command that is in progress and return to GDB command level. It's safe to type Control-C at any time because GDB doesn't allow it to take effect until it's safe. If your program is running, typing Control-C will interrupt the program and return you to the GDB prompt.

Specifying Files to Debug

GDB needs to know the file name of the program to be debugged. To debug a core dump of a previous run, GDB must be told the file name of the core dump.

The simplest way to specify the executable and core dump file names is with two command arguments given when you start GDB. The first argument is used as the file for execution and symbols, and the second argument (if any) is used as the core dump file name. Thus,

```
gdb progm core
```

specifies **progm** as the executable program and **core** as a core dump file to examine. (You don't need to have a core dump file if you plan to debug the program interactively.)

If you need to specify more precisely the files to debugged, you can do so with the following command-line options. All the options and command line arguments given are processed in sequential order. The order makes a difference when the **-x** command is used.

-symbol

```
-symbol file
-s file
```

Read symbol table from *file*.

-exec

```
-exec file
-e file
```

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

-se

```
-se file
```

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

-core

```
-core file
-c file
```

Use *file* as a core dump file to examine.

-c

```
-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

```
-command file
-x file
```

Execute GDB commands from *file*.

-directory

```
-directory directory
-d directory
```

Add *directory* to the path to search for source files.

-readnow

```
-readnow
-r
```

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

Specifying GDB Modes

The following additional command-line options can be used to affect certain aspects of the behavior of GDB:

-nx | -n

```
-nx
-n
```

Don't execute commands from the **.gdbinit** init files. Normally, the commands in these files are executed after all the command options and arguments have been processed. (See the section "Command Files" for more information.)

-q

```
-q
```

Quiet. Don't print the usual introductory messages. These messages are also suppressed in batch mode.

-batch

```
-batch
```

Run in batch mode. Exit with status 0 after processing all the command files specified with **-x** (and **.gdbinit**, if not inhibited). 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; to make this more useful, the message "Program exited normally" is not issued when running in batch mode.

-cd

```
-cd directory
```

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

-fullname | -f

```
-fullname
-f
```

This option is used when Emacs runs GDB as a subprocess. It tells GDB to produce the full file name and line number each time a stack frame is displayed (which includes each time the program stops).

-tty

```
-tty device
```

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

Editing GDB Commands

GDB provides a history buffer that stores previously executed commands. You can call any of these commands back to the command line for editing and reexecution. For example, by pressing the up-arrow key repeatedly, you can step back through each of the commands that were issued since the beginning of the session; the down-arrow key steps forward through the history buffer.

Expansion of Variable, Function, and Method Names

GDB supports command-line expansion of variable, function and method names. Type Esc-Esc or Tab to expand the current word on the command line to a matching name. If there is more than one match, the unique part is expanded and a beep occurs. To display all possible completions, type Tab again or type Esc-l.

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 allow word completion in this situation, you may enclose words in single quote marks in GDB commands. Single quotes are commonly needed in typing the name of a C++ function.

History Substitution in Commands

GDB supports the **csh** history substitution mechanism. For example, **!foo** retrieves the last command you typed that begins with **foo**. History substitution is supported across **gdb** sessions by writing the command history to a **.gdb_history** file in the current directory. Automatic creation of this history file can be disabled with the command:

```
set history save off
```

History substitution can be controlled with the **set history filename**, **set history size**, **set history save**, and **set history expansion** commands. Also see the section on history substitution in the **csh**(1) Rhapsody manual page for more information.

Emacs Command-Line Editing

You can use standard Emacs editing commands to edit the contents of the command line. All the basic Emacs command sequences work, as well as the arrow keys. The left and right arrow keys move the cursor along the command line, and the up and down arrow keys take you backward and forward through the command history.

The following list of Emacs commands shows the default key combination associated with each command and a description of what that command does.

Insertion-Point Motion Commands

Command	Description
Control-B	Move back one character
Control-F	Move forward one character
Esc b	Move back one word
Esc f	Move forward one word
Control-A	Move to beginning of line
Control-E	Move to end of line

Deletion and Restoration Commands

Command	Description
Control-D	Delete current character
Delete or Control-H	Delete previous character
Esc d	Delete current word
Esc Delete	Delete previous word
Control-K	Kill forward to end of line
Control-W	Kill region
Control-Y	Restore previous kill from buffer
Esc Y	Rotate the kill ring and yank the new top

Search Commands

Command	Description
Control-S	Search forward
Control-R	Search backward
Esc	Exit search mode

History Commands

Command	Description
Esc <	Move to beginning of history file
Esc >	Move to end of history file
Control-N	Go to next history file entry
Control-P	Go to previous history file entry

Miscellaneous Commands

Control-_	Undo the last edit.
Control-C	Interrupt a program or cancel command
Control-L	Clear screen
Control-Q	Insert a literal character
Esc Tab	Insert a Tab
Control-T	Transpose characters
Esc T	Transpose words
Control-Z	Suspend debugger, return to shell
Control-@	Set mark

Most of these commands are self-explanatory; the ones requiring more discussion are presented below.

Both delete commands and kill commands erase characters from the command line. Text that's erased by a kill key (Control-K or Control-W) is placed in the "kill buffer." If you want to restore this text, use the "yank" command, Control-Y. The yank command inserts the restored text at the current insertion point. In contrast, text that's erased by one of the delete commands (Control-D, Control-H, Esc d, and Esc h) isn't placed in the kill buffer, so it can't be restored by the yank command.

To enter a character that would otherwise be interpreted as an editing command, you must precede it with Control-Q. For example, to enter

Control-D and have it interpreted as a literal rather than as the command to delete the current character, type:

```
Control-Q Control-D
```

Editing commands can be repeated by typing Control-U followed by a number and then the command to be repeated. For example, to delete the last 15 characters typed, enter:

```
Control-U 15 Control-H
```

If you want to suspend the operation of GDB temporarily and return to the command-line prompt, type Control-Z. To return to GDB, type **%gdb** (a variant of the shell **fg** command; for more information, see the Rhapsody manual page for **csh**(1)).

Running GDB in a GNU Emacs Buffer

You can use GNU Emacs to run GDB, as well as to view (and edit) the source files for the program you're debugging with GDB.

To use the Emacs GDB interface, give the command **Esc x gdb** in Emacs. Specify the executable file you want to debug as an argument. This command starts a GDB process as a subprocess of Emacs, with input and output through a newly created Emacs buffer. You can run more than one GDB subprocess by giving the command **Esc x gdb** more than once.

Note: If your program resides in a directory other than the current directory, it can be easy to confuse Emacs about the location of the source files, in which case the auxiliary display buffer does not appear to show your source. To avoid this problem, either start GDB from the directory where your program resides or specify an absolute file name when prompted for the **Esc x gdb** argument.

Running GDB as an Emacs subprocess is just like using GDB in a Shell or Terminal window, except for two things:

- All terminal input and output goes through the Emacs buffer. This applies both to GDB commands and their output, and to the input and output done by the program you're debugging. You can copy the text of previous commands and use them again; you can even use parts of the output in this way (all the facilities of Emacs's Shell mode are available for this purpose).
- GDB displays source code through Emacs. Each time GDB displays a stack frame, Emacs automatically finds the source file for that frame and puts an arrow (**=>**) at the left margin of the current line.

Explicit GDB **list** or search commands still produce output as usual, but you'll probably have no reason to use them.

If you accidentally delete the source-display buffer, an easy way to get it back is to type the command **f** in the GDB buffer, to request a frame display; when you run under Emacs, this recreates the source buffer if necessary to show you the context of the current frame.

The source files displayed in Emacs are in ordinary Emacs buffers which are visiting the source files in the usual way. You can edit the files in these buffers if you wish; but keep in mind that GDB communicates with Emacs in terms of the line numbers as they were at compile time. If you add or delete lines from the text, the line numbers that GDB knows will no longer correspond properly to the code.

In any source file, the Emacs command Control-X space (**gdb-break**) tells GDB to set a breakpoint at the source line the point is on.

You can use these special Emacs commands in the GDB buffer:

Esc s

Execute to another source line, like the GDB **step** command.

Esc n

Execute to the next source line in this function, skipping all function calls, like the GDB **next** command.

Esc i

Execute one instruction, like the GDB **stepi** command.

Esc x gdb-nexti

Execute to next instruction, like the GDB **nexti** command.

Esc u

Move up one stack frame (and display that frame's source file in Emacs), like the GDB **up** command.

Esc d

Move down one stack frame (and display that frame's source file in Emacs), like the GDB **down** command. (You can't use **Esc d** to delete words in the usual fashion in the GDB buffer.)

Control-C Control-F
Execute until exit from the selected stack frame, like the GDB **finish** command.

Esc c
Continue execution of program, like the GDB **continue command**.

Control-h m
Describe the features of Emacs's GDB mode.

Control-x &
Insert the number in which the cursor is positioned at the end of the GDB I/O buffer. For example, if you wish to disassemble code around and address that was displayed earlier, type **disassemble**, then move the cursor to the address display and pick up the argument for **disassemble** by typing this command.

You can customize further by defining elements of the list **gdb-print-command**: once it is defined, you can format or otherwise process numbers picked up by **Control-x &** before they are inserted. A number argument to **Control-x &** indicates that you wish special formatting and also acts as an index to pick an element of the list. If the list element isa string, the number to be inserted is formatted using the Emacs function **format**: otherwise the number is passed as an argument to the corresponding list element.

Startup Files

At startup, GDB reads configuration information from startup files in the following order:

1. **/usr/lib/.gdbinit** (an Apple-provided startup file)
2. **~/.gdbinit** (your home directory startup file)
3. **./.gdbinit** (the current directory's startup file)

To make your own customizations to GDB, put GDB commands in your home directory's **.gdbinit** startup file. To make further customizations required for any specific project, put commands in a **.gdbinit** startup file within that project's directory. The startup files aren't executed if you use the **-nx** option.

For more information about making customizations to GDB, see the section "Defining and Executing Sequences of Commands" later in this chapter.

GDB Commands for Specifying and Examining Files

Usually you specify the files for GDB to work with by giving arguments when you invoke GDB. But occasionally it's necessary to change to a different file during a GDB session. Or you may run GDB and forget to specify the files you want to use. In these situations the GDB commands to specify new files are useful.

While file-specifying commands allow both absolute and relative file names as arguments, GDB always converts the file name to an absolute one and remembers it that way.

add-file

```
add-file [file] [address]
```

Adds symbols from executable file *file* to the symbol table.

add-module

```
add-module address
```

Add the object file at address *address*.

core-file

```
core-file [ file ]
```

Specify a core dump file to be used as the contents of memory. Note that the core dump contains only the writable parts of memory; the read-only parts must come from the executable file. **core-file** with no argument specifies that no core file is to be used.

This command has been superseded by the **target core** and **detach** commands.

info files

```
info files
```

Print the names of the executable and core dump files currently in use by GDB, and the file from which symbols were loaded.

kill

```
kill
```

Cancel running the program under GDB. This could be used if you want to debug a core dump instead. GDB ignores any core dump file if it's actually running the program, so the **kill** command is the only sure way to go back to using the core dump file.

load

```
load file
```

Dynamically load *file* into the running program, and record its symbols for access from GDB.

path

```
path path
```

Add one or more directories to the beginning of the search path for executable files. **$cwd** in the path means the current working directory. This path is like the **$PATH** shell variable; it is a list of directories, separated by colons. These directories are searched to find fully linked executable files and separately compiled object files as needed.

update-files

```
update-files [file]
```

Rereads symbols from file *file*. Use this if a symbol file has change since you started executing your program.

Running Your Program under GDB

To start your program under GDB, use the **run** command. The program must already have been specified with an argument to the **gdb** command (see the section "Specifying Files to Debug"); what **run** does is create an inferior process, load the program into it, and set it in motion.

The execution of a program is affected by certain types of information it receives from its superior. GDB provides ways to specify these, which you must do before starting the program. (You can change them after starting the program, but such changes don't affect the program unless you start it over again.) The types of information are:

Information	Description
The arguments	You specify the arguments to give the program by passing them as arguments to the **run** command. You can also use the **set args** command.
The environment	The 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 will be given to the program.

Information	Description
The working directory	The program inherits its working directory from GDB. You can set GDB's 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.

After the **run** command, the debugger does nothing but wait for your program to stop. See the section "Stopping and Continuing" for more information.

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 Arguments

You specify the arguments to give the program by passing them as arguments to the **run** command. They're first passed to a shell, which expands wildcard characters and performs redirection of I/O, and then passed to the program.

The **run** command with no arguments uses the same arguments used by the previous **run**.

With the **set args** command you can specify the arguments to be used the next time the program is run. If **set args** has no arguments, it means to use no arguments the next time the program is run. If you've run your program with arguments and want to run it again with no arguments, this is the only way to do so.

Your Program's Environment

Your program's 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're inherited by all the other programs you run. When debugging, it can be useful to try running the program with different environments without having to start the debugger over again.

set environment

```
set environment varname value
```

Set the environment variable *varname* to *value* (for your program only, not for GDB itself). *value* may be any string; any interpretation is supplied by your program itself.

unset environment

```
unset environment varname
```

Cancel the variable *varname* from the environment passed to your program (thereby making the variable not be defined at all, which is different from giving the variable an empty value). This doesn't affect the program until the next **run** command.

Your Program's Working Directory

Each time you start your program with **run**, the program inherits its working directory from the current working directory of GDB. GDB's working directory is initially whatever it inherited from its superior, but you can specify the working directory for GDB with the **cd** command.

The GDB working directory also serves as a default for the commands that specify files for GDB to operate on. See the section "Specifying Files to Debug."

cd

```
cd dir
```

Set the working directory for GDB and the program being debugged to *dir*. The change doesn't take effect for the program being debugged until the next time it is started.

pwd

```
pwd
```

Print GDB's working directory.

Your Program's Input and Output

By default, the program you run under GDB uses as its source of input and output the same terminal that GDB uses. GDB switches 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.

You can redirect the program's input and/or output using standard redirection commands with the **run** command. For example,

```
run > outfile
```

starts the program, diverting its output to the file **outfile**.

Another way to specify what the program should use as its source of input and output is with the **tty** command. This command accepts a file name as its argument, and causes that file to be the default for future **run** commands. For example,

```
tty /dev/ttyb
```

causes processes started with subsequent **run** commands to default to using the terminal **/dev/ttyb** as their source of input and output. An explicit redirection in **run** overrides the **tty** command.

When you use the **tty** command or redirect input in the **run** command, the input for your program comes from the specified file, but the input for GDB still comes from your terminal. The program's controlling terminal is your terminal, not the terminal that the program is reading from; so if you want to type Control-C to stop the program, you must type it on your (GDB's) terminal. Control-C typed on the program's terminal is available to the program as ordinary input.

info terminal

```
info terminal
```

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

Debugging an Already Running Process

The Mach operating system allows GDB to begin debugging an already running process that was started outside GDB. To do this you must use the **attach** command instead of the **run** command.

The **attach** command requires one argument, which is the process ID of the process you want to debug.

The first thing GDB does after arranging to debug the 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, use the **cont** (continue) command after attaching.

When you're finished debugging the attached process, you can use the **detach** command to detach the debugger from the attached process and resume execution of the process (or you can use Control-C to interrupt the process). After you give the **detach** command, that process and GDB become completely independent, and you're ready to **attach** another process or start one with **run**.

If you exit GDB or use the **run** command while you have an attached process, you kill that process. You'll be asked for confirmation if you try to do either of these things.

attach

```
attach [ arg ]
```

Attach to a process or file outside of GDB. This command attaches to another target, of the same type as your last **target** command (**info files** will show your target stack). The command may take as argument a process id or a device file. (The usual way to find out the process ID of the process is with the **ps** utility.) For a process ID, you must have permission to send the process a signal, and it must have the same effective uid as the debugger. When using **attach**, you should use the **file** command to specify the program running in the process, and to load its symbol table.

detach

```
detach
```

Detach a process or file previously attached. If a process, it is no longer traced, and it continues its execution. If you were debugging a file, the file is closed and GDB no longer accesses it.

The following commands are for connecting to a target machine or process.

target

```
target [ args ]
```

Connect to a target machine or process. The first argument is the type or protocol of the target machine. Remaining arguments are interpreted by the target protocol. For more information on the arguments for a particular protocol, type **help target** followed by the protocol name.

target child

```
target child
```

Child process (started by the **run** command).

target core

```
target core file
```

Use a core file as a target. Specify the file name of the core file.

The following commands are for kernel debugging.

kattach

kattach *hostname*

Attach to a kernel on a remote host.

kreboot

```
kreboot args
```

Reboot an attached kernel.

Stopping and Continuing

When you run a program normally, it runs until exiting. The purpose of using a debugger is so that you can stop it before that point, or so that if the program runs into trouble you can find out why.

Signals

A signal is an asynchronous event that can happen in a program. The operating system defines the possible kinds of signals, and gives each kind a name and a number. For example, SIGINT is the signal a program gets when you type Control-C; SIGSEGV is the signal a program gets from referencing a place in memory far away from all the areas in use; SIGALRM occurs when the alarm clock timer goes off (which happens only if the program has requested an alarm).

Some signals, including SIGALRM, are a normal part of the functioning of the program. Others, such as SIGSEGV, indicate errors; these signals are fatal (that is, they kill the program immediately) if the program hasn't specified in advance some other way to handle the signal. SIGINT doesn't indicate an error in the program, but it's normally fatal, so it can carry out the purpose of Control-C: to kill the program.

GDB can detect any occurrence of a signal in the program running under GDB's control. You can tell GDB in advance what to do for each kind of signal.

Normally, GDB is set up to ignore non-erroneous signals like SIGALRM (so as not to interfere with their role in the functioning of the program) but to stop the program immediately whenever an error signal happens. You can change these settings with the **handle** command. You must specify which signal you're talking about with its number.

When a signal has been set to stop the program, the program can't see the signal until you continue. It will see the signal then, if **pass** is in effect for the signal in question at that time. In other words, after GDB reports a signal, you can use the **handle** command with **pass** or **nopass** to control whether that signal will be seen by the program when you later continue it.

You can also use the **signal** command to prevent the program from seeing a signal, to cause it to see a signal it normally wouldn't see, or to give it any signal at any time. See the section "Continuing" below.

info signals

```
info signals [ signalnum ]
```

Print a table of all the kinds of signals and how GDB has been told to handle each one. You can use this to see the signal numbers of all the defined types of signals. Specify a signal number in order to print information about that signal only.

handle

```
handle signalnum keywords
```

Change the way GDB handles signal *signalnum*. The *keywords* say what change to make.

The keywords allowed by the **handle** command can be abbreviated. Their full names are:

Keyword	Description
stop	GDB should stop the program when this signal happens. This implies the **print** keyword as well.
print	GDB should print a message when this signal happens.
nostop	GDB shouldn't stop the program when this signal happens. It may still print a message telling you that the signal has come in.
noprint	GDB shouldn't mention the occurrence of the signal at all. This implies the **nostop** keyword as well.
pass	GDB should allow the program to see this signal; the program will be able to handle the signal, or may be terminated if the signal is fatal and not handled.
nopass	GDB shouldn't allow the program to see this signal.

Breakpoints

A breakpoint makes your program stop whenever a certain point in the program is reached. You set breakpoints explicitly with GDB commands, specifying the place where the program should stop by line number, function name, or exact address in the program. You can add various other conditions to control whether the program will stop.

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

Each breakpoint is assigned a number when it's created; these numbers are successive integers starting with 1. 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 the program until you enable it again.

The **info breakpoints** command prints a list of all breakpoints set and not cleared, showing their numbers, their location in the program, and any special features in use for them. Disabled breakpoints are included in the list, but marked as disabled. **info breakpoints** with a breakpoint number as its argument lists only that breakpoint. The convenience variable **$_** and the default address for the **x** command are set to the address of the last breakpoint listed (see the section "Examining Memory"). The **info breakpoints** command can be abbreviated as **info break**.

Breakpoints can't be used in a program if any other process is running that program. Attempting to run or continue the program with a breakpoint in this case will cause GDB to stop it. When this happens, you must remove or disable the breakpoints, and then continue.

Setting Breakpoints

Breakpoints are set with the **break** command (abbreviated **b**). There are several ways to specify where the breakpoint should go:

GDB allows you to set any number of breakpoints at the same place in the program. This can be useful when the breakpoints are conditional (see the section "Break Conditions").

break

```
break function
```

Set a breakpoint at entry to *function*. You can also set a breakpoint at the entry to a method, as described in the section "Method Names in Commands."

```
break linenum
```

Set a breakpoint at *linenum* in the current source file (the last file whose source text was printed). This breakpoint will stop the program just before it executes any of the code from that line.

```
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 file:linenum
```

Set a breakpoint at *linenum* in *file*.

```
break file:function
```

Set a breakpoint at entry to *function* found in *file*. Specifying a file name as well as a function name is superfluous except when multiple files contain identically named functions. This doesn't work for Objective-C methods; see the section "Method Names in Commands" for information on setting breakpoints for methods.

```
break *address
```

Set a breakpoint at *address*. You can use this to set breakpoints in parts of the program that don't have debugging information or source files.

```
break
```

Set a breakpoint at the next instruction to be executed in the selected stack frame (see the 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 the **finish** command in the frame inside of 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

```
break [args]  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. *args* stands for one of the possible arguments described above (or no argument) specifying where to break. See the section "Break Conditions" for more information.

tbreak

```
tbreak [args]
```

Set a breakpoint enabled only for one stop. *args* are the same as in the **break** command, and the breakpoint is set in the same way, but the breakpoint is automatically deleted the first time it's hit.

rbreak *regex*

```
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.

future-break

```
future-break function
```

Set a breakpoint at *function*, and defer the breakpoint if *function*'s address can't be resolved. Use this command to set breakpoints in code that has not been loaded yet (for example, code in a bundle or dynamically shared library). As files are loaded, GDB checks their symbols to see if any deferred breakpoints can be resolved. If a breakpoint can be resolved, it becomes enabled. If a future breakpoint can never be resolved, it stays in the breakpoint list until you explicitly delete it. Note that if you spell the function name wrong, the breakpoint will never be resolved and you will receive no error message. This command only applies to the Mach version of GDB.

Watchpoints

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

Watchpoints currently execute two orders of magnitude more slowly that other breakpoints, but this can be well worth it to catch errors where you have no clue what part of your program is the culprit.

watch

```
watch expr
```

Set a watchpoint for this expression.

info watchpoints

```
info watchpoints
```

Print a list of watchpoints and breakpoints; it is the same as **info break**.

Warning: In multithreaded 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.

Clearing Breakpoints

It's often necessary to eliminate a breakpoint once it has done its job and you no longer want the program to stop there. This is called clearing (or deleting) the breakpoint. A breakpoint that has been cleared no longer exists in any sense.

With the **clear** command you can clear breakpoints according to where they are in the program. With the **delete** command you can clear individual breakpoints by specifying their breakpoint numbers.

It isn't necessary to clear a breakpoint to proceed past it. GDB automatically ignores breakpoints in the first instruction to be executed when you continue execution at the same address where the program stopped.

clear

```
clear
```

Clear any breakpoints at the next instruction to be executed in the selected stack frame (see the section "Selecting a Frame"). When the innermost frame is selected, this is a good way to clear a breakpoint that the program just stopped at.

```
clear function
clear file:function
```

Clear any breakpoints set at entry to the *function*.

```
clear linenum
clear file:linenum
```

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

delete [breakpoints] [*bnum* ...]

```
delete [breakpoints] [bnum ...]
```

Clear the breakpoints whose numbers are specified as arguments. If no argument is specified, delete all breakpoints (GDB asks confirmation unless you have **set confirm off**). A deleted breakpoint is forgotten completely.

Disabling Breakpoints

Rather than clearing a breakpoint, you might prefer to disable it. This makes the breakpoint inoperative as if it had been cleared, but remembers the information about the breakpoint so that you can enable it again later.

You enable and disable breakpoints with the **enable** and **disable** commands, specifying one or more breakpoint numbers as arguments. Use **info breakpoints** to print a list of breakpoints if you don't know which breakpoint numbers to use.

A breakpoint can have any of four states of enablement:

- Disabled. The breakpoint has no effect on the program.
- Enabled. The breakpoint will stop the program. A breakpoint made with the **break** command starts out in this state.
- Enabled once. The breakpoint will stop the program, but when it does so it will become disabled.
- Enabled for deletion. The breakpoint will stop the program, but immediately afterward it is deleted permanently. A breakpoint made with the **tbreak** command starts out in this state.

You can enable and disable breakpoints with the following commands:

enable

```
enable [breakpoints] bnum ...
```

Enable the specified breakpoints (or all defined breakpoints). They become effective once again in stopping the program, until you specify otherwise.

enable once

```
enable  [breakpoints] once bnum ...
```

Enable the specified breakpoints temporarily. Each will remain enabled only until the next time it stops the program (unless you use one of these commands to specify a different state before that time comes). Also see the **tbreak** command, which sets a breakpoint and enables it once.

enable delete

```
enable  [breakpoints] delete bnum ...
```

Enable the specified breakpoints to work once and then die. Each of the breakpoints will be deleted the next time it stops the program (unless you use one of these commands to specify a different state before that time comes).

disable

```
disable  [breakpoints] bnum ...
```

Disable the specified breakpoints. A disabled breakpoint has no effect but isn't forgotten. All options such as ignore counts, conditions, and commands are remembered in case the breakpoint is enabled again later.

Aside from the automatic disablement or deletion of a breakpoint when it stops the program, which happens only in certain states, the state of enablement of a breakpoint changes only when one of the above commands is used (except if the breakpoint is set with **tbreak**).

Break Conditions

The simplest sort of breakpoint breaks every time the program reaches a specified place. You can also specify a condition for a breakpoint. A condition is simply a Boolean expression. A breakpoint with a condition evaluates the expression each time the program reaches it, and the program stops only if the condition is true.

This is the converse of using assertions for program validation; in that situation, you want to stop when the assertion is violated—that is, when the condition is false. In C, if you want to test an assertion expressed by the condition *assert*, you should set the condition !*assert* on the appropriate breakpoint.

Break conditions may have side effects, and may even call functions in your program. This can be useful, for example, to activate functions that log program process or to use your own print functions to format special data structure. The effects are completely predictable unless there's another enabled breakpoint at the same address. (In that case, GDB might see the other breakpoint first and stop the program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible than break conditions for the purpose of performing side effects when a breakpoint is reached (see the section "Executing Commands at a Breakpoint").

Break conditions can be specified when a breakpoint is set, by using **if** in the arguments to the **break** command (see the section "Setting Breakpoints"). They can also be changed at any time with the **condition** command:

condition

```
condition bnum expression
```

Specify *expression* as the break condition for breakpoint number *bnum*. From now on, this breakpoint will stop the program only if the value of *expression* is true (nonzero, in C). 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, however.

```
condition bnum
```

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

Ignoring breakpoints

A special feature is provided for one kind of condition: to prevent the breakpoint from doing anything until it has been reached a certain number of times. This is done with the "ignore count" of the breakpoint. When the program reaches a breakpoint whose ignore count is positive, then instead of stopping, it just decrements the ignore count by 1 and continues.

If a breakpoint has a positive ignore count and a condition, the condition isn't checked. Once the ignore count reaches 0, the condition will start to be checked.

You could achieve the effect of the ignore count with a condition such as **$foo--<= 0** using a debugger convenience variable that's decremented each time. That's why the ignore count is considered a special case of a condition. See the section "Convenience Variables."

ignore

```
ignore bnum count
```

Set the ignore count of breakpoint number *bnum* to *count*. The next *count* times the breakpoint is reached, it won't stop.

To make the breakpoint stop the next time it's reached, specify a count of 0.

continue

```
continue n
```

Continue execution of the program, setting the ignore count of the breakpoint that the program stopped at to *n* minus 1. Continuing through the breakpoint doesn't itself count as one of *n*. Thus, the program won't stop at this breakpoint until the *n*th time it's hit.

This command is allowed only when the program stopped due to a breakpoint. At other times, the argument to **cont** is ignored.

Executing Commands at a Breakpoint

With the **commands** *bnum* command, you can give the breakpoint *bnum* a series of commands to execute when the program stops due to that breakpoint. For example, you might want to print the values of certain expressions, or enable other breakpoints. The commands themselves appear on the following lines. **if** and **while** statements are allowed inside the commands list. Type a line containing just **end** to terminate the commands.

Breakpoint commands can be used to start up the program again. Simply use the **continue** command, or **step**, or any other command that resumes execution. However, any remaining breakpoint commands are ignored. When the program stops again, GDB will act according to why that stop took place.

If the first command specified is **silent**, the usual message about stopping at a breakpoint isn't printed. This may be desirable for breakpoints that are to print a specific message and then continue. If the remaining commands also print nothing, you'll see no sign that the breakpoint was reached at all. **silent** isn't really a command; it's meaningful only at the beginning of the commands for a breakpoint.

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

Here's how you could use breakpoint commands to print the value of **x** at entry to **foo** whenever it's positive. We assume that the newly created breakpoint is number 4; **break** will print the number that's assigned.

```
break foo if x>0
commands 4
silent
printf "x is %d\n",x
cont
end
```

or

```
break foo
commands 4
silent
if (x > 0)
  printf "x is %d\n",x
  end
cont
end
```

One application for breakpoint commands is to correct one bug so you can test 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 **cont** command so that the program doesn't stop, and start with the **silent** command so that no output is produced. Here's an example:

```
break 403
commands 5
silent
set x = y + 4
cont
end
```

One deficiency in the operation of breakpoints that continue automatically appears when your program uses raw mode for the terminal. GDB reverts to its own terminal modes (not raw) before executing commands, and then must switch back to raw mode when your program is continued. This causes any pending terminal input to be lost.

You could get around this problem by putting the actions in the breakpoint condition instead of in commands. For example,

```
condition 5  (x = y + 4), 0
```

is a condition expression that will change **x** as needed, then always have the value 0 so the program won't stop. Loss of input is avoided here because break conditions are evaluated without changing the terminal modes. When you want to have nontrivial conditions for performing the side effects, the operators **&&**, **||**, and **?:** may be useful.

commands

`commands` *bnum*

Specify commands for breakpoint number *bnum*. The commands themselves appear on the following lines. **if** and **while** statements are allowed inside the commands list. Type a line containing just **end** to terminate the commands.

To remove all commands from a breakpoint, use the command **commands** and follow it immediately by **end**; that is, give no commands.

Breakpoint Menus

In Objective-C and C++, classes can use the same names for their methods or member functions. This is called overloading. When a function name or method name is overloaded, **break** *function* is not enough to tell GDB where you want a breakpoint. In this instance, GDB offers you a menu of numbered choices for different possible breakpoints and waits for your selection.

Continuing

After your program stops, most likely you'll want it to run some more if the bug you're looking for hasn't happened yet. You can do this with the **continue** command:

If the program stopped at a breakpoint, the place to continue running is the address of the breakpoint. You might expect that continuing would just stop at the same breakpoint immediately. In fact, **continue** takes special care to prevent that from happening. You don't need to clear the breakpoint to proceed through it after stopping at it.

You can, however, specify an ignore count for the breakpoint that the program stopped at, by means of an argument to the **continue** command. See the section "Break Conditions" above.

If the program stopped because of a signal other than SIGINT or SIGTRAP, continuing will cause the program to see that signal. You may not want this to happen. For example, if the program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but the program would probably terminate immediately as a result of the fatal signal once it sees the signal. To prevent this, you can continue with **signal 0**. You can also act in advance to prevent the program from seeing certain kinds of signals, using the **handle** command (see the section "Signals").

You can use **fg** as a synonym for **continue**.

continue

```
continue
```

Continue running the program at the place where it stopped.

Stepping

Stepping means setting your program in motion for a limited time, so that control will return automatically to the debugger after one line of code or one machine instruction. Breakpoints are active during stepping and the program will stop for them even if it hasn't gone as far as the stepping command specifies.

A typical technique for using stepping is to put a breakpoint at the beginning of the function or the section of the program in which a problem is believed to lie, and then step through the suspect area examining interesting variables until the problem happens.

The **cont** command can be used after stepping to resume execution until the next breakpoint or signal.

step

```
step [count]
```

Continue running the program until control reaches a different line, then stop it and return to the debugger. If an argument is specified, proceed as in **step**, but do so *count* times. If a breakpoint or a signal not related to stepping is reached before *count* steps, stepping stops right away. You can abbreviate this command as **s**.

next

```
next [count]
```

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. An argument is a repeat count, as in **step**. You can abbreviate this command as **n**.

finish

```
finish
```

Continue running until just after the selected stack frame returns (or until there's some other reason to stop, such as a fatal signal or a breakpoint). Upon return, the value returned is printed and put in the value history. Contrast this with the **return** command, described in the section "Returning from a Function."

until

```
until
```

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 through 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.

```
until linenum
```

Continue running until line number *linenum* is reached or the current stack frame returns. This is equivalent to setting a breakpoint at *linenum*, executing a **finish** command, and deleting the breakpoint. This form of the command uses breakpoints and hence is quicker than **until** without an argument.

stepi

```
stepi [count]
```

Execute one machine instruction, then stop and return to the debugger. It's often useful to do **display/i $pc** when stepping by machine instructions. This will cause the next instruction to be executed to be displayed automatically at each stop (see the section "Automatic Display"). An argument is a repeat count, as in **step**. You can abbreviate this command as **si**.

nexti

```
nexti [count]
```

Proceed one machine instruction, but if it's a subroutine call, proceed until the subroutine returns. An argument is a repeat count, as in **next**. You can abbreviate this command as **ni**.

Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

Each time your program performs a function call, the information about where in the program the call was made from is saved in a block of data called a *stack frame*. The frame also contains the arguments of the call and the local variables of the function that was called. All the stack frames are allocated in a region of

memory called the call stack. When your program stops, the GDB commands for examining the stack allow you to see all this information.

Stack Frames

The call stack is divided into contiguous pieces called frames; each frame is the data associated with one call to one function. The frame contains the arguments given to the function, the function's local variables, and the address at which the function is executing.

When your program is started, the stack has only one frame, that of the function **main()**. This is called the initial frame, or the outermost frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the innermost frame. This is the most recently created of all the stack frames that still exist.

Inside your program, stack frames are identified by their addresses. A stack frame consists of many bytes, each of which has its own address; each kind of computer has a convention for choosing the address of one of those bytes to serve as the address of the frame. Usually this address is kept in a register called the frame pointer register while execution is going on in that frame.

GDB assigns numbers to all existing stack frames, starting with 0 for the innermost frame, 1 for the frame that called it, and so on upward. These numbers don't really exist in your program; they simply give you a way of talking about stack frames in GDB commands.

At any given time, one of the stack frames is selected by GDB; many GDB commands refer implicitly to this selected frame. In particular, whenever you ask GDB for the value of a variable in the program, the value is found in the selected frame. You can select any frame using the **frame**, **up**, and **down** commands; subsequent commands will operate on that frame.

When the program stops, GDB automatically selects the currently executing frame and describes it briefly, as the **frame** command does (see the section "Information about a Frame").

Some compilers provide a way to compile functions so that they operate without stack frames. (For example, the GCC option **-fomit-frame-pointer** generates functions without a frame.) This is occasionally done with heavily used library functions to save the frame setup time. GDB has limited facilities for dealing with these function invocations. If the innermost

function invocation has no stack frame, GDB nevertheless regards it as though it had a separate frame, which is numbered zero as usual, allowing correct tracing of the function call chain. However, GDB has no provision for frameless functions elsewhere in the stack.

Backtraces

A backtrace is a summary of how the program got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame 0) followed by its caller (frame 1), and on up the stack.

Each line in a backtrace shows the frame number, the program counter, the function and its arguments, and the source file name and line number (if known). For example:

```
(gdb) backtrace
#0  0x3eb6 in fflush ()
#1  0x24b0 in _fwalk ()
#2  0x2500 in _cleanup ()
#3  0x2312 in exit ()
```

backtrace [*n*]

Print a backtrace of the entire stack: one line per frame for all frames in the stack. You can stop the backtrace at any time by typing the system interrupt character, normally Control-C. With a positive argument, the command prints the innermost *n* frames; with a negative argument, it prints the outermost *n* frames. You can abbreviate this command as **bt**. Two aliases for this command are **where** and **info stack**.

Selecting a Frame

Most commands for examining the stack and other data in the program work on whichever stack frame is selected at the moment. Below are the commands for selecting a stack frame.

All these commands (except **up-silently** and **down-silently**) end by printing some information about the frame that has been selected: the frame number, the function name, the arguments, the source file and line number of execution in that frame, and the text of that source line. For example:

```
#3  main (argc=3, argv=??, env=??) at main.c, line 67
67      read_input_file (argv[i]);
```

After such a printout, the **list** command with no arguments will print ten lines centered on the point of execution in the frame. See the section “Printing Source Lines.”

frame

```
frame n
```

Select and print frame number *n*. Recall that frame 0 is the innermost (currently executing) frame, frame 1 is the frame that called the innermost one, and so on. The highest-numbered frame is **main**'s frame.

```
frame addr
```

Select and print the frame at address *addr*. This is useful mainly if the chaining of stack frames has been damaged by a bug, making it impossible for GDB to assign numbers properly to all frames. In addition, this can be useful if the program has multiple stacks and switches between them.

up

```
up n
```

Select and print the frame *n* frames up from the frame previously selected. For positive numbers *n*, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. *n* defaults to 1.

up-silently *n*

```
up-silently n
```

Same as the **up** command, but doesn't print anything (this is useful in command scripts).

down *n*

```
down n
```

Select and print the frame *n* frames down from the frame previously selected. For positive numbers *n*, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. *n* defaults to 1.

down-silently

```
down-silently n
```

Same as the **down** command, but doesn't print anything (this is useful in command scripts).

Information about a Frame

There are several other commands to print information about the selected stack frame.

frame

```
frame [n]
```

This command prints a brief description of the selected stack frame. With an argument, this command is used to select a stack frame (the argument can be a stack frame number or the address of a frame); with no argument, it doesn't change which frame is selected, but still prints the same information. You can abbreviate this command as **f**.

info frame

```
info frame
```

This command prints a verbose description of the selected stack frame, including the address of the frame, the addresses of the next frame down (called by this frame) and the next frame up (caller of this frame), the address of the frame's arguments, the program counter saved in it (the address of execution in the caller frame), and which registers were saved in the frame. The verbose description is useful when something has gone wrong that has made the stack format fail to fit the usual conventions.

info frame *addr*

```
info frame addr
```

Print a verbose description of the frame at address *addr*, without selecting that frame. The selected frame remains unchanged by this command.

info args

```
info args
```

Print the arguments of the selected frame, each on a separate line.

info locals

```
info locals
```

Print the local variables of the selected frame, each on a separate line.

Examining Source Files

GDB knows which source files your program was compiled from, and can print parts of their text. When your program stops, GDB spontaneously prints the line it stopped in. Likewise, when you select a stack frame (see the section "Selecting a Frame"), GDB prints the line in which execution in that frame has stopped. You can also print parts of source files by explicit command.

Viewing Files in Project Builder

To be able to dynamically open and view source files in Project Builder, use the view command.

view

```
view [ host ]
```

Cause source files to be viewed in Project Builder, either on the local machine or on a remote *host*.

unview

```
unview
```

Cause source files not to be viewed in Project Builder.

Printing Source Lines

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

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

list

```
list linenum
```

Print lines centered around *linenum* in the current source file.

```
list function
```

Print lines centered around the beginning of *function*.

```
list
```

Print 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 (see the section "Examining the Stack"), this prints lines centered around that line.

You can repeat a **list** command by pressing the Return key; however, any argument that was used is discarded, so this is equivalent to typing simply **list**. An exception is made for an argument of -; that argument is preserved in repetition so that each repetition moves up in the 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. The possible arguments for **list** are as follows:

Command	Description
list ,*last*	Print lines ending with *last*.
list *first***,**	Print lines starting with *first*.
list +	Print lines just after the lines last printed.

Command	Description
list –	Print lines just before the lines last printed.
list *linespec*	Print lines centered around the line specified by *linespec* (described below).
list *first*,*last*	Print lines from *first* to *last*. Both arguments are *linespec*s.

Here are the possible ways to specify a value for *linespec*:

Value	Description
linenum	Specifies line *linenum* of the current source file. When a **list** command has two *linespec*s, 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, this specifies the line *offset* lines down from the first *linespec*.
–*offset*	Specifies the line *offset* lines before the last line printed.
file:*linenum*	Specifies line *linenum* in the source file *file*.
function	Specifies the line of the left brace ({) that begins the body of *function*.
file:*function*	Specifies the line of the left brace ({) that begins the body of *function* in *file*. The file name is needed with a function name only for disambiguating identically named functions in different source files.
*****addr	Specifies the line containing the program address *addr*. *addr* may be any expression.

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

```
set listsize count
```

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

show listsize

```
show listsize
```

Display the number of lines that **list** prints.

The **info line** command is used to map source lines to program addresses:

info line

```
info line [ line ]
```

Print the starting and ending addresses of the compiled code for source line *line*, which can be specified as:

Option	Description
linenum	list around that line in current file,
file:linenum,	list around that line in that file,
function,	list around beginning of that function, or
file:function,	distinguish among like-named static functions.

With no argument, the command describes the last source line that was listed.

The default address for the **x** command is changed to the starting address of the line, so that **x/i** is sufficient to begin examining the machine code (see the section "Examining Memory"). Also, this address is saved as the value of the convenience variable **$_** (see the section "Convenience Variables").

Searching Source Files

The **forward-search** command (or its alias, **search**) and the **reverse-search** command are useful when you want to locate text within the current source file.

forward-search

```
forward-search regexp
search regexp
```

This command checks each line, starting with the one following the last line listed, for a match for *regexp*, which must be a regular expression (see the Rhapsody manual page for **ed**). It lists the line that's found. You can abbreviate this command as **fo**.

reverse-search

```
reverse-search regexp
```

The command checks each line, starting with the one before the last line listed and going backward, for a match for *regexp*. It lists the line that's found. You can abbreviate this command as **rev**.

Specifying Source Directories

Executable programs sometimes don't record the directories of the source files they were compiled from, just the names. Even when they do, the directories could be moved between the compilation and your debugging

session. GDB remembers a list of pathnames of directories in which it will search for source files; this list is called the source path (note that GDB doesn't use the environment variable PATH to search for source files). Each time GDB wants a source file, it tries each directory in the list, starting from the beginning, until it finds a file with the desired name.

When you start GDB, its source path is set to **$cdir:$cwd** (the current working directory, and the directory in which the source file was compiled into object code). To add other directories, use the **directory** command:

directory

```
directory dirname
```

Add directory with the pathname *dirname* to the beginning of the source path. Several directory names may be given to this command separated by a colon or whitespace. You may specify a directory that is already in the source path; this move it forward so GDB searches it sooner.

```
directory
```

Reset the source path to **$cdir:$cwd**, the default.This requires confirmation.

Examining Data

The most common way to examine data in your program is with the **print** command (abbreviated **p**) or its synonym **inspect:**

Another way to examine data is with the **x** command (see "Examining Memory" below). It examines data in memory at a specified address and prints it in a specified format.

If you are interested in information about types or about how the fields of a struct or class are declared, use the **ptype** command rather than **print.**

print

```
print exp
```

This command evaluates and prints the value of any valid expression of the language the program is written in (currently, C, C++, and Objective-C). Variables accessible are those of the lexical environment of the selected stack frame, plus all those whose scope is global or an entire file.

exp is any valid expression, and the value of *exp* is printed in a format appropriate to its data type. To print data in another format, you can cast *exp* to the desired type or use the **x** command.

$*num* gets previous value number *num*. **$** and **$$** are the last two values. **$$***num* refers to the *num*'th value back from the last one. Names starting with **$** refer to registers (with the values they would have if the program were to return to the stack frame now selected, restoring all registers saved by frames farther in) or else to debugger convenience variables (any such name that isn't a known register). Use assignment expressions to give values to convenience variables.

{*type*}*adrexp* refers to a datum of data type *type*, located at address *adrexp*. **@** is a binary operator for treating consecutive data objects anywhere in memory as an array. *foo***@***num* gives an array whose first element is *foo*, whose second element is stored in the space following where *foo* is stored, etc. *foo* must be an expression whose value resides in memory.

exp may be preceded with */fmt*, where *fmt* is a format letter but no count or size letter (see the description of the **x** command).

print-object

```
print-object object
```

Print *object* by sending **description** to it. *object* must be an Objective-C object. You can abbreviate this command as **po**.

set

```
set exp
```

The **set** command works like the **print** command, except that the expression's value isn't displayed. This is useful for modifying the state of your program. For example:

```
set x=3
set close_all_files()
```

Expressions

Many different GDB commands accept an expression and compute its value. Any kind of constant, variable, or operator defined by the programming language you're using is legal in an expression in GDB. This includes conditional expressions, function calls, casts, and string constants. It unfortunately does not include symbols defined by preprocessor **#define** constants.

GDB supports three kinds of operators in addition to those of programming languages:

Operator	Description
file-or-function::*variable-name*	:: allows you to specify a variable in terms of the file or function it's defined in.
@	@ is a binary operator for treating parts of memory as arrays. See the section "Artificial Arrays" below for more information.
{*type*} *addr*	Refers to an object of type *type* stored at address *addr* in memory. *addr* may be any expression whose value is an integer or pointer (but parentheses are required around nonunary operators, just as in a cast). This construct is allowed no matter what kind of data is officially supposed to reside at *addr*.

Program Variables

The most common kind of expression to use is the name of a variable in your program.

Variables in expressions are understood in the selected stack frame (see the section "Selecting a Frame"); they must be either global (or static) or visible according to the scope rules of the programming language from the point of execution in that frame. This means that in the function

```
foo (a)
    int a;
{
    bar (a);
    {
        int b = test ();
        bar (b);
    }
}
```

the variable **a** is usable whenever the program is executing within the function **foo()**, but the variable **b** is usable only while the program is executing inside the block in which **b** is declared.

There is an exception: you can refer to a variable or function whose scope is a single source file even if the current execution point is not in this file. But it is possible to have more than one such variable or function with the same name (in different source files). If that happens, referring to that name has unpredictable effects. If you wish, you can specify a static variable in a particular function or file, using the colon-colon notation:

```
file::variable
function::variable
```

Here *file* or *function* is the name of the context for the static variable. In the case of file names, you can use quotes to make sure GDB parses the file name as a single word—for example, to print a global value of **x** defined in **f2.c**:

```
(gdb) p 'f2.c'::x
```

This use of colon-colon is very rarely in conflict with the very similar use of the same notation in C++. GDB also supports use of the C++ scope resolution operator in GDB expressions.

Warning: Occasionally, a local variable may appear to have the wrong value at certain points in a function—just after entry to a new scope, and just before exit. You may see this problem when you are stepping by machine instructions. This is because on most machines it takes more than one instruction to set up a stack frame (including local variable definitions); if you are stepping by machine instructions, variables may appear to have the wrong values until the stack frame is completely built. On exit, it usually also takes more than one machine instruction to destroy a stack frame; after you begin stepping through that group of instructions, local variable definitions may be gone.

Artificial Arrays

It's often useful to print out several successive objects of the same type in memory (for example, a section of an array, or an array of dynamically determined size for which only a pointer exists in the program).

This can be done by constructing an "artificial array" with the binary operator **@**. The left operand of **@** should be the first element of the desired array, as an individual object. The right operand should be the length of the array. The result is an array value whose elements are all of the type of the left argument. The first element is actually the left argument; the second element comes from bytes of memory immediately following those that hold the first element, and so on. For example, if a program says

```
int *array = (int *) malloc (len * sizeof (int));
```

you can print the contents of **array** with

```
p *array@len
```

The left operand of **@** must reside in memory. Array values made with **@** in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions.

Sometimes the artificial array mechanism is not quite enough; in moderately complex data structure, the elements of interest may not actually be adjacent—for example, if you are interested in the values of pointers in an array. One useful work-around in this situation is to use a convenience variable as a counter in an expression that prints the first interesting value and then repeat that expression using a carriage return. For instance, suppose you have an array **dtab** of pointers to structures, and you are interested in the values of a field **fv** in each structure. Here is an example of what you might type:

```
set $i=0
p dtab[$i++]->fv
<CR>
<CR>
```

Output Formats

GDB normally prints all values according to their data types. Sometimes this isn't what you want. For example, you might want to print a number in hexadecimal, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or an instruction. These things can be done with output formats.

The simplest use of output formats is to specify how to print a value already computed. This is done by starting the arguments of the **print** command with a slash and a format letter. The format letters supported are:

Letter	Description
x	Regard the bits of the value as an integer, and print the integer in hexadecimal.
d	Print as integer in signed decimal.
u	Print as integer in unsigned decimal.
o	Print as integer in octal.
t	Print as integer in binary.
a	Print as an address, both absolute in hexadecimal and then relative to a symbol defined at an address below it.
c	Regard as an integer and print as a character constant.
f	Regard the bits of the value as a floating-point number and print using typical floating-point syntax.

For example, to print the program counter in hexadecimal (see the section "Registers"), type

```
p/x $pc
```

No space is required before the slash because command names in GDB can't contain a slash.

To reprint the last value in the value history with a different format, you can use the **print** command with just a format and no expression. For example, **p/x** reprints the last value in hexadecimal.

Examining Memory

The command **x** (for "examine") can be used to examine memory under explicit control of formats, without reference to the program's data types.

x is followed by a slash and an output format specification, followed by an expression for an address:

```
x/nfu addr
```

The expression *addr* doesn't need to have a pointer value (though it may); it's used as an integer, as the address of a byte of memory.

n, *f*, and *u* are all optional parameters that specify how much memory to display and how to format it; *addr* is an expression giving the address where you want to start displaying memory. If you use the defaults for *nfu*, you need not type the slash. Several commands set convenient defaults for *addr*.

Letter	Description
n,	The repeat count is a decimal integer; the default is 1. It specifies how much memory (counting by units *u*) to display.
f,	The display format is one of the formats used by print, or **s** (null-terminated string) or **i** (machine instruction). The default is **x** (hexadecimal) initially, or the format from the last time you used **print** or **x**.
u	These letters specify the size of unit to examine: **b** Examine individual bytes. **h** Examine halfwords (two bytes each). **w** Examine words (four bytes each). **g** Examine giant words (eight bytes).

If neither the manner of printing nor the size of unit is specified, the default is the same as was used last. If you don't want to use any letters after the slash, you can omit the slash as well.

You can also omit the address to examine. Then the address used is just after the last unit examined. This is why string and instruction formats actually compute a unit-size based on the data: so that the next string or instruction examined will start in the right place. The **print** command sometimes sets the default address for the **x** command; when the value

printed resides in memory, the default is set to examine the same location. **info line** also sets the default for **x** to the address of the start of the machine code for the specified line and **info breakpoints** sets it to the address of the last breakpoint listed.

When you repeat an **x** command by pressing the Return key, the address specified previously (if any) is ignored; instead, the command examines successive locations in memory rather than the same one.

You can examine several consecutive units of memory with one command by writing a repeat count after the slash (before the format letters, if any). The repeat count must be a decimal integer. It has the same effect as repeating the **x** command that many times except that the output may be more compact with several units per line.

```
x/10i $pc
```

Prints ten instructions starting with the one to be executed next in the selected frame. After doing this, you could print another ten following instructions with

```
x/10
```

in which the format and address are allowed to default.

The addresses and contents printed by the **x** command aren't put in the value history because there's often too much of them and they would get in the way. Instead, GDB makes these values available for subsequent use in expressions as values of the convenience variables **$_** and **$__** (that is, **$** followed by one or two underscores).

After an **x** command, the last address examined is available for use in expressions in the convenience variable **$_**. The contents of that address, as examined, are available in the convenience variable **$__**.

If the **x** command has a repeat count, the address and contents saved are from the last memory unit printed; this isn't the same as the last address printed if several units were printed on the last line of output.

Automatic Display

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the "automatic display list" so that GDB will print its value each time the program stops. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

```
2: foo = 38
3: bar[5] = (struct hack *) 0x3804
```

showing item numbers, expressions, and their current values.

display

```
display exp
```

Add the expression *exp* to the list of expressions to display each time the program stops.

```
display/fmt exp
```

Add the expression *exp* to the automatic display list, and display it in the format *fmt*. *fmt* should specify only a display format, not a size or count.

```
display/fmt addr
```

Add the expression *addr* as a memory address to be examined each time the program stops. *fmt* should be either **i** or **s**, or it should include a unit size or a number of units. See the section "Examining Memory."

```
display
```

Display the current values of the expressions on the list, just as is done when the program stops.

undisplay

```
undisplay [ n ... ]
delete display [ arg ... ]
```

Remove item number *n* from the list of expressions to display. With no argument, cancels all automatic-display expressions.

info display

```
info display
```

Print the list of expressions to display automatically, each one with its item number, but without showing the values.

enable display

```
enable display [ arg ... ]
```

Enable some expressions to be displayed when the program stops. Arguments are the code numbers of the expressions to resume displaying. No argument means enable all automatic-display expressions.

disable display

```
disable display [ arg ... ]
```

Disable some expressions to be displayed when the program stops. Arguments are the code numbers of the expressions to stop displaying. No argument means disable all automatic-display expressions.

Value History

Every value printed by the **print** command is saved for the entire session in GDB's "value history" so that you can refer to it in other expressions.

The values printed are given "history numbers" for you to refer to them by. These are successive integers starting with 1. **print** shows you the history number assigned to a value by printing **$*n* =** before the value, where *n* is the history number.

To refer to any previous value, use **$** followed by the value's history number. The output printed by **print** is designed to remind you of this. **$** alone refers to the most recent value in the history, and **$$** refers to the value before that.

For example, suppose you have just printed a pointer to a structure and want to see the contents of the structure. It's enough to type

```
p *$
```

If you have a chain of structures where the component **next** points to the next one, you can print the contents of the next one with

```
p *$.next
```

It might be useful to repeat this command many times by pressing the Return key.

Note that the history records values, not expressions. If the value of **x** is 4 and you type

```
print x
set x=5
```

then the value recorded in the value history by the **print** command remains 4 even though **x**'s value has changed.

Convenience Variables

GDB provides "convenience variables" that you can use within GDB to hold a value for future reference. These variables exist entirely within GDB; they aren't part of your program, and setting a convenience variable has no effect on further execution of your program. That's why you can use them freely.

Convenience variables have names starting with **$**. Any name starting with **$** can be used for a convenience variable, unless it's one of the predefined set of register names (see the section "Registers").

You can save a value in a convenience variable with an assignment expression, just as you would set a variable in your program. For example:

```
set $foo = *object_ptr
```

would save in **$foo** the value contained in the object pointed to by **object_ptr**.

Convenience variables don't need to be explicitly declared; using a convenience variable for the first time creates it. However, its value is **void** until you assign it a value. You can alter the value with another assignment at any time.

Convenience variables have no fixed types. You can assign a convenience variable any type of value, even if it already has a value of a different type. The convenience variable as an expression has whatever type its current value has.

One way to use a convenience variable is as a counter to be incremented or a pointer to be advanced. For example:

```
set $i = 0
print bar[$i++]->contents
    repeat that command by typing RET.
```

Some convenience variables are created automatically by GDB and given values likely to be useful.

Variable	Description
$_	The variable **$_** (single underscore) is automatically set by the **x** command to the last address examined (see the section "Examining Memory"). Other commands which provide a default address for **x** to examine also set **$_** to that address; these commands include **info line** and **info breakpoint**.
$__	The variable **$__** (two underscores) is automatically set by the **x** command to the value found in the last address examined.

Registers

Machine register contents can be referred to in expressions as variables with names starting with **$**.

The names **$pc** and **$sp** are used for the program counter register and the stack pointer. **$fp** is used for a register that contains a pointer to the current stack frame. To see a list of all the registers, use the command **info registers**.

Some registers have distinct “raw” and “virtual” data formats. This means that the data format in which the register contents are saved by the operating system isn’t the same one that your program normally sees. For example, the registers of the 68882 floating-point coprocessor are always saved in “extended” format, but all C programs expect to work with “double” format. In such cases, GDB normally works with the virtual format only (the format that makes sense for your program), but the **info registers** command prints the data in both formats.

Register values are relative to the selected stack frame (see the section “Selecting a Frame”). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the real contents of all registers, you must select the innermost frame (with **frame 0**).

Some registers are never saved (typically those numbered 0 or 1) because they’re used for returning function values; for these registers, relativization makes no difference.

For example, you could print the program counter in hexadecimal with

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add 4 to the stack pointer with

```
set $sp += 4
```

The last is a way of removing one word from the stack. This assumes that the innermost stack frame is selected. Setting **$sp** isn’t allowed when other stack frames are selected.

info registers

```
info registers [regname]
```

With no argument, print the names and relativized values of all registers except floating-point registers. With an argument, print the relativized value of register *regname*. *regname* may be any register name valid on the machine you’re using, with or without the initial **$**.

info all-registers

```
info all-registers
```

Print the names and values of all registers, including floating-point registers.

Miscellaneous Commands

call

```
call arg
```

Call a function in the inferior process. The argument is the function name and arguments, in standard C notation. The result is printed and saved in the value history, if it isn't void.

disassemble

```
disassemble [ arg [ arg ] ]
```

Disassemble a specified section of memory. The default is the function surrounding the **pc** of the selected frame. With a single argument, the function surrounding that address is dumped. Two arguments are taken as a range of memory to dump.

Examining the Symbol Table

The commands described in this section allow you to make inquiries for information about the symbols (names of variables, functions, and types) defined in your program. GDB finds this information in the symbol table contained in the executable file; it's inherent in the text of your program and doesn't change as the program executes.

Occasionally, you may need to refer to symbols that contain unusual characters, which GDB ordinarily treats as word delimiters. The most frequent case is in referring to static variables in other source files. File names are recorded in object files as debugging symbols, but GDB would ordinarily parse a typical file name, like **foo.c** as three words "**foo**", ".", and "**c**". To allow GDB to recognize foo.c as a single symbol, enclose it in single quotes; for example **p 'foo.c'::x** looks up the value of **x** in the scope of the file **foo.c**.

whatis

```
whatis [exp]
```

With no argument, print the data type of **$**, the last value in the value history. With an argument, print the data type of expression *exp*. *exp* isn't actually evaluated, and any operations inside it that have side effects (such as assignments or function calls) don't take place.

info address

```
info address symbol
```

Describe where the data for *symbol* is stored. For register variables, this says which register. For other automatic variables, this prints the stack-frame offset at which the variable is always stored. Note the contrast with **print &***symbol*, which doesn't work at all for register variables, and which for automatic variables prints the exact address of the current instantiation of the variable.

info functions

```
info functions [regexp]
```

With no argument, print the names and data types of all defined functions. With an argument, print the names and data types of all defined functions whose names contain a match for regular expression *regexp* (for information about regular expressions, see the Rhapsody manual page for **ed**). For example, **info fun step** finds all functions whose names include **step**; **info fun ^step** finds those whose names start with **step**.

info source

```
info source
```

Show the name of the current source file—that is, the source file for the function containing the current point of execution—and the language it was written in.

info sources

```
info sources
```

Print the names of all source files in the program for which there is debugging information, organized into two lists: files whose symbols have already been read, and files whose symbols will be read when needed.

info types

```
info types [regexp]
```

With no argument, print all data types that are defined in the program. With an argument, print all data types that are defined in the program whose names contain a match for regular expression *regexp*.

This command differs from **ptype** in two ways: first, like **whatis**, it does not print a detailed description; second, it lists all source files where a type is defined.

info variables

```
info variables [regexp]
```

With no argument, print the names and data types of all top-level variables that are declared outside functions. With an argument, print the names and data

types of all variables declared outside functions, whose names contain a match for regular expression *regexp*.

ptype

```
ptype typename
```

Print a description of data type *typename*. *typename* may be the name of a type, or for C code it may have the form **class** *class-name* **struct** *struct-tag*, **union** *union-tag* or **enum** *enum-tag*. The selected stack frame's lexical context is used to look up the name.

```
ptype [exp]
```

Print a description of the type of expression *exp*. **ptype** differs from **whatis** by printing a detailed description, instead of just the name of the type.

Setting Variables

set

```
set
```

Perform an assignment *var* **=** *exp*. You must type the **=**. *var* may be a debugger convenience variable (a name starting with **$**), a register (one of a few standard names starting with **$**), or an actual variable in the program being debugged. *exp* is any expression. Use **set variable** for variables with names identical to **set** subcommands.

With a subcommand listed below, the **set** command modifies parts of the GDB environment (you can see these environment settings with **show** and its subcommands). In general, use **on** (or no argument) to enable a feature, and **off** to disable it.

set args

```
set args arg ...
```

Set arguments to give the program being debugged when it is started. Follow this command with any number of arguments to be passed to the program.

set autoload-breakpoints

```
set autoload-breakpoints on/off
```

Set automatic resetting of breakpoints in dynamic code.

set autoload-symbols

```
set autoload-symbols on/off
```

Set automatic loading of symbols of dynamic code.

set catch-user-commands-errors

```
set catch-user-commands-errors on/off
```

Set whether to ignore errors in user commands.

set complaints

```
set complaints num
```

Set the maximum number of complaints about incorrect symbols.

set confirm

```
set confirm on/off
```

Set whether to confirm potentially dangerous operations.

set demangle-style

```
set demangle-style on/off
```

Set the current C++ demangling style.

set editing

```
set editing on/off
```

Set command-line editing.

set environment

```
set environment var value
```

Set environment variable and value to give the program. Arguments are *var value* where *var* is the variable name and *value* is the value. Values of environment variables are uninterpreted strings. This command does not affect the program until the next **run** command.

set force_cpluplus

```
set force_cpluplus on/off
```

Set if you know better than debugger about C++.

set history expansion

```
set history expansion on/off
```

Set history expansion on command input.

set history filename

```
set history filename file
```

Set the filename in which to record the command history (the list of previous commands of which a record is kept).

set history ignoredups

```
set history ignoredups on/off
```

Set whether history condenses sequences of identical commands.

set history save

```
set history save on/off
```

Set whether the history record is saved when you exit **gdb**.

set history size

```
set history size size
```

Set the size of the command history (the number of previous commands to keep a record of).

set input-radix

```
set input-radix num
```

Set the default input radix for entering values.

set language

```
set language lang
```

Set the language to be used in debugging.

set lazy-read

```
set lazy-read on/off
```

Set whether inferior's memory is read lazily.

set listsize

```
set listsize num
```

Set the number of source lines GDB will print by default with **list**.

set output-radix

```
set output-radix num
```

Set the default output radix for print values.

set print address

```
set print address on/off
```

Set printing of addresses.

set print array

```
set print array on/off
```

Set pretty printing of arrays.

set print asm-demangle

```
set print asm-demangle on/off
```

Set demangling of C++ names in disassembly listings.

set print demangle

```
set print demangle on/off
```

Set demangling of encoded C++ names when displaying symbols.

set print elements

```
set print elements size
```

Set limit on string chars or array elements to print. The value **0** causes there to be no limit.

set print max-symbolic-offset

```
set print max-symbolic-offset max-offset
```

Set the largest offset that will be printed in *symbol*+1234 form.

set print null-stop

```
set print null-stop on/off
```

Set printing of character arrays to stop at first null character.

set print object

```
set print object on/off
```

Set printing of object's derived type based on vtable info.

set print pretty

```
set print pretty on/off
```

Set pretty printing of structures.

set print repeats

```
set print repeats size
```

Set threshold for repeated print elements.

set print sevenbit-strings

```
set print sevenbit-strings on/off
```

Set printing of 8-bit characters in strings as *nnn*.

set print symbol-filename

```
set print symbol-filename on/off
```

Set printing of file name and line number with symbols.

set print union

```
set print union on/off
```

Set printing of unions interior to structures.

set print vtbl

```
set print vtbl on/off
```

Set printing of C++ virtual function tables.

set prompt

```
set prompt string
```

Set GDB's prompt. The argument is an unquoted string.

set radix

```
set radix on/off
```

Set the default input and output number radix.

set symbol-reloading

```
set symbol-reloading on/off
```

Set dynamic symbol table reloading multiple times in one run.

set verbose

```
set verbose on/off
```

Set whether verbose printing of informational messages is enabled or disabled.

set view-host

```
set view-host host
```

Set the host to connect to when viewing.

set view-program

```
set view-program name
```

Set the name of the program to connect to when viewing.

set variable

```
set variable var = exp
```

Same as **set**; use **set variable** in cases where *var* is identical to one of the **set** subcommands.

Status Inquiries

info address

```
info address var
```

Describe where the specified variable is stored.

info all-registers

```
info all-registers
```

List of all registers, including floating-point registers, and their contents.

info args

```
info args
```

Provide information about the argument variables of the current stack frame.

info breakpoints

```
info breakpoints [ num ]
```

Provide information about the status of all breakpoints, or of breakpoint number *num*. The second column displays **y** for enabled breakpoints, **n** for disabled, **o** for enabled once (disable when hit), or **d** for enabled but delete when hit. The address and the file/line number are also displayed.

The convenience variable **$_** and the default examine address for **x** are set to the address of the last breakpoint listed. The convenience variable **$bpnum** contains the number of the last breakpoint set.

info classes

```
info classes
```

Show all Objective-C classes (Mach only).

info copying

```
info copying
```

Show conditions for redistributing copies of GDB.

info display

```
info display
```

Show expressions to display when program stops, with code numbers.

info files

```
info files
```

Show the names of targets and files being debugged. Shows the entire stack of targets currently in use (including the exec-file, core-file, and process, if any), as well as the symbol file name.

info float

```
info float
```

Show the status of the floating point unit.

info frame

```
info frame [ addr ]
```

Provide information about the selected stack frame, or the frame at *addr*.

info handle

```
info handle
```

Show what debugger does when program gets various signals.

info functions

```
info functions [ regexp ]
```

Show all function names, or those matching *regexp*.

info line

```
info line [ line_spec ]
```

Core addresses of the code for a source line. *line_spec* can be specified as

Specifier	Description
linenum	list around that line in current file,
file:*linenum*	list around that line in that file,
function	list around beginning of that function, or
file:*function*	distinguish among like-named static functions.

The default is to describe the last source line that was listed.

This sets the default address for **x** to the line's first instruction so that **x/i** suffices to start examining the machine code. The address is also stored as the value of **$_**.

info locals

```
info locals
```

Provide information about the local variables of the current stack frame.

info program

```
info program
```

Show the execution status of the program.

info registers

```
info registers [ register_name ]
```

Show a list of registers and their contents for the selected stack frame. A register name as argument means describe only that register.

info selectors

```
info selectors
```

Show all Objective-C selectors (Mach only).

info set

info set

Show all GDB settings.

info signals

```
info signals [ sig_num ]
```

Show what GDB does when the program gets various signals. Specify a signal number to print information about that signal only.

info sources

```
info sources
```

Show the names of source files in the program.

info source

```
info source
```

Provide information about the current source file.

info stack

```
info stack [ count ]
```

Provide a backtrace of the stack, or of the innermost *count* frames.

info target

```
info target
```

Same as **info files**.

info terminal

```
info terminal
```

Print inferior's saved terminal status.

info types

```
info types [ regexp ]
```

Show all type names, or those matching *regexp*.

info variables

```
info variables [ regexp ]
```

Show all global and static variable names, or those matching *regexp*.

info warranty

```
info warranty
```

Show information pertaining to warranty.

info watchpoints

```
info watchpoints [ num ]
```

Provide information about the status of all watchpoints, or of watchpoint number *num*. The second column displays **y** for enabled watchpoints or **n** for disabled ones.

show autoload-breakpoints

```
show autoload-breakpoints
```

Show automatic resetting of breakpoints in dynamic code.

show autoload-symbols

```
show autoload-symbols
```

Show automatic loading of symbols of dynamic code.

show args

```
show args
```

Show arguments to give program being debugged when it is started.

show catch-user-commands-errors

```
show catch-user-commands-errors
```

Show whether to ignore errors in user commands.

show commands

```
show commands
```

Show the status of the command editor.

show complaints

```
show complaints
```

Show the maximum number of complaints about incorrect symbols.

show copying

```
show copying
```

Show conditions for redistributing copies of GDB.

show confirm

```
show confirm
```

Show whether to confirm potentially dangerous operations.

show convenience

```
show convenience
```

Show the debugger convenience variables. These variables are created when you assign them values; thus, p**rint $foo=1** gives **$foo** the value 1. Values may be of any type.

A few convenience variables are given values automatically: **$_** holds the last address examined with **x** or **info lines**, and **$__** holds the contents of the last address examined with **x**.

show demangle-style

```
show demangle-style on/off
```

Show the current C++ demangling style.

show directories

```
show directories
```

Current search path for finding source files. **$cwd** in the path means the current working directory. **$cdir** in the path means the compilation directory of the source file.

show editing

```
show editing
```

Show command-line editing.

show environment

```
show environment [ var ]
```

Show the environment to give the program, or one variable's value. With an argument *var*, prints the value of environment variable *var* to give the program

being debugged. With no arguments, prints the entire environment to be given to the program.

show force_cplusplus

```
show force_cplusplus
```

Show if you know better than the debugger about C++.

show history expansion

```
show history expansion
```

Show history expansion on command input.

show history filename

```
show history filename
```

Show the filename in which to record the command history (the list of previous commands of which a record is kept).

show history ignoredups

```
show history ignoredups
```

Show whether history condenses sequences of identical commands.

show history save

```
show history save
```

Show saving of the history record on exit.

show history size

```
show history size
```

Show the size of the command history (that is, the number of previous commands to keep a record of).

show input-radix

```
show input-radix
```

Show the default input radix for entering values.

show language

```
show language
```

Show the programming language being used in debugging.

show lazy-read

```
show lazy-read
```

Show if inferior's memory is read lazily.

show listsize

```
show listsize
```

Show the number of lines printed by **list** with no argument.

show output-radix

```
show output-radix num
```

Show the default output radix for print values.

show paths

```
show paths
```

Show the current search path for finding object files. **$cwd** in the path means the current working directory. This path is like the **$PATH** shell variable; that is, a list of directories separated by colons. These directories are searched to find fully linked executable files and separately compiled object files as needed.

show print address

```
show print address
```

Show printing of addresses.

show print array

```
show print array
```

Show prettyprinting of arrays.

show print asm-demangle

```
show print asm-demangle
```

Show demangling of C++ names in disassembly listings.

show print demangle

```
show print demangle
```

Show demangling of encoded C++ names when displaying symbols.

show print elements

```
show print elements
```

Show limit on string chars or array elements to print.

show print max-symbolic-offset

```
show print max-symbolic-offset
```

Show the largest offset that will be printed in *symbol*+1234 form.

show print null-stop

```
show print null-stop
```

Show printing of character arrays to stop at first null character.

show print object

```
show print object
```

Show printing of object's derived type based on vtable info.

show print pretty

```
show print pretty
```

Show pretty printing of structures.

show print repeats

```
show print repeats
```

Show threshold for repeated print elements.

show print sevenbit-strings

```
show print sevenbit-strings
```

Show printing of 8-bit characters in strings as *nnn*.

show print symbol-filename

```
show print symbol-filename
```

Show printing of file name and line number with symbols.

show print union

```
show print union
```

Show printing of unions interior to structures.

show print vtbl

```
show print vtbl
```

Show printing of C++ virtual function tables.

show prompt

```
show prompt
```

Show GDB's prompt.

show radix

```
show radix
```

Show the default input and output number radix.

show symbol-reloading

```
show symbol-reloading
```

Show if dynamic symbol table reloads multiple times in one run.

show values

```
show values [ idx ]
```

Elements of value history around item number *idx* (or last ten).

show verbose

```
show verbose
```

Show whether verbosity is on or off.

show version

```
show version
```

Report what version of GDB this is.

show view-host

```
show view-host
```

Show host to connect to when viewing.

show view-program

```
show view-program
```

Show name of program to connect to when viewing.

show user

```
show user
```

Show definitions of user-defined commands.

show warranty

```
show warranty
```

Show information pertaining to warranty.

Debugging PostScript Code

This section describes three commands that are useful when debugging PostScript source files.

These commands aren't built-in commands; rather, the OPENSTEP environment defines them in a system **.gdbinit** file located in the directory **/usr/lib**.

This file is read when you start running GDB (the contents of this file are shown later in this chapter).

showps, shownops

```
showps
shownops
```

The **showps** and **shownops** commands turn on and off (respectively) the display of PostScript code being sent from your application to the Window Server. Your application must be running before you can issue either of these commands.

flushps

```
flushps
```

The **flushps** command sends pending PostScript code to the Window Server. This command lets you flush the application's output buffer, causing any PostScript code waiting there to be interpreted immediately. Your application must be running before you can issue this command.

traceevents

```
traceevents
```

Trace PostScript events. When an event is queued, it is logged to standard error.

tracenoevents

```
tracenoevents
```

Turn off tracing of PostScript events.

waitps

```
waitps
```

Wait until the DPS context's destination is ready to receive more input.

Debugging Objective-C Code

This section provides information about some commands and command options that are useful for debugging Objective-C code.

Setting the Language

The syntax accepted by certain GDB commands, such as **break**, is determined by the programming lanugage being debugged. By default, the

language is set to C, so you can always use C syntax in GDB commands. On Mach, when the language is set to C, GDB also accepts Objective-C syntax (for example, the use of colons in method names and the message-sending syntax) . In the PDO version of GDB, the language must be set to Objective-C for the debugger to be able to accept Objective-C syntax in commands.

GDB tries to set the language it accepts in its commands according to the language that the program being debugged uses. If the program's source files have the extension **.m** or **.M**, then GDB assumes that the program is written in Objective-C and sets the language it accepts accordingly. The **show language** command displays what the language is currently set to. You can use the **set language** command to override the value. To set the language to Objective-C, enter this command:

```
set language objective c
```

The **set language** command is particularly useful if you're debugging a mixed-language program. For example, if you're stopped in a C module and you want to send a message to an Objective-C object, you won't be able to because GDB won't recognize the square bracket syntax as an Objective-C message. You must first set the language to Objective-C, then send the message.

Method Names in Commands

The following commands have been extended to accept Objective-C method names as line specifications:

```
clear
break
info line
jump
list
```

For example, to set a breakpoint at the **create** instance method of class Fruit in the program currently being debugged, enter:

```
break –[Fruit create]
```

To list ten program lines around the initialize class method, enter:

```
list +[NSText initialize]
```

In the PDO version of GDB, the plus or minus sign is required. On Mach, the plus or minus sign is optional, but you can use it to narrow the search. On Mach, it's also possible to specify just a method name:

```
break create
```

You must specify the complete method name, including any colons. If your program's source files contain more than one **create** method, you'll be presented with a numbered list of classes that implement that method. Indicate your choice by number, or type 0 to exit if none apply.

As another example, to clear a breakpoint established at the **makeKeyAndOrderFront:** method of the NSWindow class, enter:

```
clear -[NSWindow makeKeyAndOrderFront:]
```

If you're using the PDO version of GDB and you don't know the exact method name or you don't know the name of the class to which it belongs, you can use the **info functions** command to find out. (On Mach, you can use the **info selectors** command instead.) Use **info functions** followed by a regular expression to narrow the search. For example, to find out all the methods and functions that contain the string "set," enter:

```
info functions set
```

To find just the methods that contain "set," include the bracket in the regular expression. (You must escape the braket with a backslash because it is part of the regular expression syntax.)

```
info functions \[.*set
```

To find just the methods that begin with the string "set," include the space as part of the name:

```
info functions \[.* set
```

Command Descriptions

This section describes commands and options that are useful in debugging Objective-C code. Some of these are new commands that have been implemented in OPENSTEP, and some are previously existing GDB commands that have been extended in OPENSTEP.

The info Command

The **info** command takes two additional options on Mach:

info classes

```
info classes [regexp]
```

Display all Objective-C classes in your application, or those matching the regular expression *regexp*.

info selectors

```
info selectors [regexp]
```

Display all Objective-C selector names (or those matching the regular expression *regexp*), and also each selector's unique number.

If you don't limit the command's scope by entering a regular expression, the resulting listing can be quite long. To terminate a listing at any point and return to the GDB prompt, type Control-C.

Two standard **info** command options have been extended. The **info types** command recognizes and lists the Objective-C **id** type. The **info line** command recognizes Objective-C method names as line specifications.

The print Command

The **print** command has been extended to allow the evaluation of Objective-C objects and message expressions. Consider, for example, this program excerpt:

```
@implementation Fruit : NSObject
{
    char *color;
    int diameter;
}

+ create  {
    id newInstance;
    newInstance = [super new];       // creates instance of Fruit
    [ newInstance color:"green" ];   // set the color
    [ newInstance diameter:1];       // set the diameter
    return newInstance;              // return the new instance
}
. . .
@end
```

Once this code has been executed, you can use GDB to examine **newInstance** by entering:

```
print newInstance
```

The output looks something like this (of course, the address wouldn't be the same):

```
$1 = (id) 0x1a020
```

As declared, **newInstance** is a pointer to an Objective-C object. To see the structure this variable points to, enter:

```
print *newInstance
```

GDB displays:

```
$3 = {
    isa = 0x120b4;
    color = 0x26bf "green";
    diameter = 1;
}
```

This structure contains the instance variables defined above for objects of the Fruit class. It also contains a pointer, called **isa**, that points to its class object. To see the identity of this class, enter:

```
print *newInstance->isa
```

GDB displays:

```
$4 = {
    isa = 0x12090;
    super_class = 0x124a4;
    name = 0x125a2 "Fruit";
    version = 0;
    info = 17;
    instance_size = 12;
    ivars = 0x1203c;
    methods = 0x120ec;
    cache = 0x22080;
}
```

The instance variable **name** verifies that this is an instance of the Fruit class.

You can also evaluate a message expression with the **print** command. As a by-product of the evaluation, the message is sent to the receiving object. For example, the following command sets the color of the Fruit object to red:

```
print [newInstance color: "red"]
```

The set Command

The **set** command can be used to evaluate and send a message expression. For example, the following command sets the color of the Fruit object to red:

```
set [newInstance color: "red"]
```

The step Command

The **step** command has been extended to let you step through the execution of an Objective-C message. By repeatedly executing the **step** command, you can watch the chain of events that make up the execution of a message.

If you step into a message and don't want to follow the details of its execution, enter:

```
finish
```

This command completes the execution of the message and stops the program at the next statement. To avoid stepping into the message in the first place, use the **next** command rather than **step**. The **next** command instructs GDB to execute the current command and stop only when control returns to the current stack frame.

Debugging Mach Threads

The following commands have been provided in the Mach version of GDB to support the debugging of Mach threads.

thread-list

```
thread-list thread
```

List all threads that exist in the program being debugged (abbreviated **tl**).

thread-select

```
thread-select thread
```

Select a thread (abbreviated **ts**). For example, **ts 2** selects thread *2*.

Debugging Mach Core Files

OPENSTEP GDB has been extended to allow debugging of core files in the Mach-O file format. Core files are generated in the **/cores** directory, if it exists; otherwise, they're generated in the current working directory.

The **info files** command lists information about the contents of the core file. This tells you what segments of address space exist in the core file, how many threads exist in the core image, and what the program counter is for each thread. Thread 0 is selected by default, so if you do a **bt** it will apply to thread 0. The **thread-list** and **thread-select** commands, documented in the section "Debugging Mach Threads" above, work with core files. All the normal debugger commands can also be used while debugging the core image.

Altering Execution

There are several ways to alter the execution of your program with GDB commands.

Assignment to Variables

To alter the value of a variable, evaluate an assignment expression. For example:

```
print x=4
```

would store the value 4 into the variable **x**, and then print the value of the assignment expression (which is 4).

If you aren't interested in seeing the value of the assignment, use the **set** command instead of the **print** command. **set** is the same as **print** except that the expression's value isn't printed and isn't put in the value history. The expression is evaluated only for side effects.

GDB allows more implicit conversions in assignments than C does; you can freely store an integer value into a pointer variable or vice versa, and any structure can be converted to any other structure that's the same length or shorter.

All the other C assignment operators such as **+=** and **++** are supported as well.

To store into arbitrary places in memory, use the {...} construct to generate a value of specified type at a specified address. For example:

```
set {int}0x83040 = 4
```

Continuing at a Different Address

jump

```
jump linenum
```

Resume execution at line number *linenum*. Execution may stop immediately if there's a breakpoint there.

The **jump** command doesn't change the current stack frame, or the stack pointer, or the contents of any memory location or any register other than the program counter. If *linenum* is in a different function from the one currently executing, the results may be wild if the two functions expect different patterns of arguments or of local variables. For this reason, the **jump**

command requests confirmation if the specified line isn't in the function currently executing.

jump *

```
jump *address
```

Resume execution at the instruction at address *address*.

A somewhat similar effect can be obtained by storing a new value into the register **$pc**. For example:

```
set $pc = 0x485
```

specifies the address at which execution will resume, but doesn't resume execution. That doesn't happen until you use the **cont** command or a stepping command.

Giving Your Program a Signal

signal

```
signal signal
```

Resume execution where your program stopped, but immediately give it the signal *signal*. *signal* can be the name or the number of a signal. For example, on many systems **signal 2** and **signal SIGINT** are both ways of sending an interrupt signal. Alternatively, if *signal* is 0, continue execution without giving a signal. This is useful when your program stopped on account of a signal and would ordinarily see the signal when resumed with the continue command; **signal 0** causes it to resume without a signal.

Invoking the **signal** command is not the same as invoking the **kill** utility from the shell. Sending a signal with **kill** causes GDB to decide what to do with the signal depending on the signal handling tables. The signal command passes the signal directly to your program.

Returning from a Function

return

```
return [exp]
```

You can make any function call return immediately by using the **return** command.

First select the stack frame that you want to return from (see the section "Selecting a Frame"). Then type the **return** command. If you want to specify the value to be returned, give that as an argument.

The selected stack frame (and any other frames inside it) is popped, leaving its caller as the innermost remaining frame. That frame becomes selected. The specified value is stored in the registers used for returning values of functions.

The **return** command doesn't resume execution; it leaves the program stopped in the state that would exist if the function had just returned. Contrast this with the **finish** command, which resumes execution until the selected stack frame returns naturally.

Defining and Executing Sequences of Commands

GDB provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files.

User-Defined Commands

A "user-defined command" is a sequence of GDB commands to which you assign a new name as a command. This is done with the **define** command.

User-defined commands may take up to 10 arguments. Within the definition of the command, you refer to the arguments as **$arg0**, **$arg1**, and so on up to **$arg9**. For example, if you defined a command that took two arguments, you refer to the first one specified on the command line as **$arg0** and the second one as **$arg1**.

When they're executed, the commands of the definition aren't printed. An error in any command stops execution of the user-defined command.

Commands that would ask for confirmation if used interactively proceed without asking when used inside a user-defined command. Many GDB commands that normally print messages to say what they're doing omit the messages when used in a user-defined command.

define

```
define commandname
```

Define a command named *commandname*. If there's already a command by that name, you're asked to confirm that you want to redefine it.

The definition of the command is made up of other GDB command lines, which are given following the **define** command. **if** and **while** statements are

allowed within the definition. The end of the command definition is marked by a line containing just the command **end**. For example:

```
define w
    where
end
```

document

`document` *commandname*

Create documentation for the user-defined command *commandname*. The command *commandname* must already be defined. This command reads lines of documentation just as **define** reads the lines of the command definition. After the **document** command is finished, **help** on command *commandname* will print the documentation you have specified.

You may use the **document** command again to change the documentation of a command. Redefining the command with **define** doesn't change the documentation, so be sure to keep the documentation up to date.

Command Files

A command file for GDB is a file of lines that are GDB commands. Comments (lines starting with **#**) may also be included. An empty line in a command file does nothing; it doesn't cause the last command to be repeated, as it would from the terminal.

When GDB starts, it automatically executes its "init files" (command files named **.gdbinit**). GDB first reads the init file (if any) in your home directory and then the init file (if any) in the current working directory. (The init files aren't executed if the **-nx** option is given.) You can also request the execution of a command file with the **source** command:

The lines in a command file are executed sequentially. They aren't printed as they're executed. An error in any command terminates execution of the command file.

Commands that would ask for confirmation if used interactively proceed without asking when used in a command file. Many GDB commands that normally print messages to say what they're doing omit the messages when used in a command file.

source

`source` *file*

Execute the command file *file*.

Commands for Controlled Output

During the execution of a command file or a user-defined command, the only output that appears is what's explicitly printed by the commands of the definition. This section describes three additional commands useful for generating exactly the output you want.

echo

`echo` *`text`*

Print *text*. Nonprinting characters can be included in *text* using C escape sequences, such as **\n** to print a newline. No newline will be printed unless you specify one. In addition to the standard C escape sequences, a backslash followed by a space stands for a space. This is useful for display a string with space at the beginning or the end, since leading and trailing space are otherwise trimmed from all arguments.

A backslash at the end of *text* is ignored. It's useful for producing a string ending in spaces, since trailing spaces are trimmed from all arguments. A backslash at the beginning preserves leading spaces in the same way, because the escape sequence backslash-space stands for a space. Thus, to print " variable foo = ", do

```
echo \ variable foo = \
```

output

`output` *`expression`*

Print just the value of *expression*. A newline character isn't printed, and the value isn't entered in the value history.

`output/`*`fmt expression`*

Print the value of *expression* in format *fmt*. See "Output Formats" in the section "Examining Data" for more information.

printf

`printf` *`format-string, arg`* `[,` *`arg`*`] ...`

Print the values of the arguments, under the control of *format-string*. This command is identical in its operation to its C library equivalent (see the Rhapsody manual page for **printf()** for format codes). The only backslash-escape sequences that you can use in the format string are the simple ones that consist of the backslash followed by a letter.

Miscellaneous Commands

make

```
make [ args ]
```

Run the **make** program using the rest of the line as arguments.

select-frame

```
select-frame
```

Select the frame at **fp, pc**.

shell

```
shell [ command ]
```

Execute the rest of the line as a shell command. With no arguments, run an inferior shell.

Legal Considerations

Permission is granted to make and distribute verbatim copies of this chapter provided its copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this chapter under the conditions for verbatim copying, provided also that the section entitled “GDB General Public License” (below) is included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this chapter into another language, under the above conditions for modified versions, except that the section entitled “GDB General Public License” may be included in a translation approved by the author instead of in the original English.

Distribution

GNU software is *free*; this means that everyone is free to use it and free to redistribute it on a free basis. GNU software is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of GNU software that they might get from you. The precise conditions

are found in the GNU General Public License that appears following this section.

You may obtain a complete machine-readable copy of any OPENSTEP-modified source code for Free Software Foundation software under the terms of Free Software Foundation's general public licenses, without charge except for the cost of media, shipping and handling, upon written request to Technical Services at NeXT Software, Inc.

When making a request, please specify which GNU software programs you're interested in receiving. GNU programs released by NeXT currently include:

```
gcc       GNU compiler
gdb       GNU debugger
gas       GNU assembler
emacs     GNU text editor
```

If you want an unmodified, verbatim copy of any GNU software (including GNU software that's not part of the OPENSTEP software release), you can order it from the Free Software Foundation. Though GNU software itself is free, the distribution service is not. For further information, write to:

Free Software Foundation
675 Mass. Ave.
Cambridge, MA 02139

Income that Free Software Foundation derives from distribution fees goes to support the Foundation's purpose: the development of more free software to distribute.

GDB General Public License

The license agreements of most software companies keep you at the mercy of those companies. By contrast, our general public license is intended to give everyone the right to share GDB. To make sure that you get the rights we want you to have, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. Hence this license agreement.

Specifically, we want to make sure that you have the right to give away copies of GDB, that you receive source code or else can get it if you want it, that you can change GDB or use pieces of it in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of

GDB, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for GDB. If GDB is modified by someone else and passed on, we want its recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

Therefore we (Richard Stallman and the Free Software Foundation, Inc.) make the following terms which say what you must do to be allowed to distribute or change GDB.

Copying Policies

1. You may copy and distribute verbatim copies of GDB source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy a valid copyright notice "Copyright (c) 1988 Free Software Foundation, Inc." (or with whatever year is appropriate); keep intact the notices on all files that refer to this License Agreement and to the absence of any warranty; and give any other recipients of the GDB program a copy of this License Agreement along with the program. You may charge a distribution fee for the physical act of transferring a copy.

2. You may modify your copy or copies of GDB or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:

- cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and

- cause the whole of any work that you distribute or publish, that in whole or in part contains or is a derivative of GDB or any part thereof, to be licensed at no charge to all third parties on terms identical to those contained in this License Agreement (except that you may choose to grant more extensive warranty protection to some or all third parties, at your option).

- You may charge a distribution fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

  Mere aggregation of another unrelated program with this program (or its derivative) on a volume of a storage or distribution medium does not bring the other program under the scope of these terms.

3. You may copy and distribute GDB (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:

- accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,

- accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal shipping charge) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,

- accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

  For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs.

4. You may not copy, sublicense, distribute or transfer GDB except as expressly provided under this License Agreement. Any attempt otherwise to copy, sublicense, distribute or transfer GDB is void and your rights to use the program under this License agreement shall be automatically terminated. However, parties who have received computer software programs from you with this License Agreement will not have their licenses terminated so long as such parties remain in full compliance.

5. If you wish to incorporate parts of GDB into other free programs whose distribution conditions are different, write to the Free Software Foundation at 675 Mass. Ave., Cambridge, MA 02139. We have not yet worked out a simple rule that can be stated here, but we will often permit this. We will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software.

Your comments and suggestions about our licensing policies and our software are welcome! Please contact the Free Software Foundation, Inc., 675 Mass. Ave., Cambridge, MA 02139, or call (617)876-3296.

No Warranty

BECAUSE GDB IS LICENSED FREE OF CHARGE, WE PROVIDE ABSOLUTELY NO WARRANTY, TO THE EXTENT PERMITTED BY APPLICABLE STATE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, FREE SOFTWARE FOUNDATION, INC, RICHARD M. STALLMAN AND/OR OTHER PARTIES PROVIDE GDB “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF GDB IS WITH YOU. SHOULD GDB PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW WILL RICHARD M. STALLMAN, THE FREE SOFTWARE FOUNDATION, INC., AND/OR ANY OTHER PARTY WHO MAY MODIFY AND REDISTRIBUTE GDB AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS) GDB, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.