

Debugging with GDB
The GNU Source-Level Debugger
Edition 4.12, for GDB version 4.16.1 January 1994"
Richard M. Stallman and Roland H. Pesch

Send bugs and comments on GDB to bug-gdb@prep.ai.mit.edu.

Debugging with GDB

Copyright © 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995 Free Software Foundation, Inc.

Published by the Free Software Foundation
59 Temple Place - Suite 330,
Boston, MA 02111-1307 USA
Printed copies are available for \$20 each.
ISBN 1-882114-11-6

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

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also 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 manual into another language, under the above conditions for modified versions.

Summary of GDB

The purpose of a debugger such as GDB is to allow you to see what is going on “inside” another program while it executes—or what another program was doing at the moment it crashed.

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

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

You can use GDB to debug programs written in C or C++. For more information, see *C and C++*.

CONTENTS

Summary of GDB	1
1. A Sample GDB Session	2
2. Getting In and Out of GDB	5
2.1 Invoking GDB	5
2.2 Quitting GDB	7
2.3 Shell commands	7
3. GDB Commands	9
3.1 Command syntax	9
3.2 Command completion	9
3.3 Getting help	10
4. Running Programs Under GDB	13
4.1 Compiling for debugging	13
4.2 Starting your program	13
4.3 Your program's arguments	14
4.4 Your program's environment	14
4.5 Your program's working directory	15
4.6 Your program's input and output	15
4.7 Debugging an already-running process	16
4.8 Killing the child process	16
4.9 Additional process information	17
4.10 Debugging programs with multiple threads	17
4.11 Debugging programs with multiple processes	19
5. Stopping and Continuing	20
5.1 Breakpoints, watchpoints, and exceptions	20
5.2 Continuing and stepping	28
5.3 Signals	30
5.4 Stopping and starting multi-thread programs	31
6. Examining the Stack	33
6.1 Stack frames	33
6.2 Backtraces	34
6.3 Selecting a frame	34
6.4 Information about a frame	35
7. Examining Source Files	37
7.1 Printing source lines	37
7.2 Searching source files	38
7.3 Specifying source directories	38
7.4 Source and machine code	39
8. Examining Data	41
8.1 Expressions	41
8.2 Program variables	41
8.3 Artificial arrays	42
8.4 Output formats	43
8.5 Examining memory	44
8.6 Automatic display	45
8.7 Print settings	46
8.8	

Value history	50
8.9 Convenience variables	51
8.10 Registers	52
9. C Language Support	54
9.1 C operators	54
9.2 C and C++ constants	55
9.3 C++ expressions	55
9.4 C and C++ defaults	56
9.5 GDB and C	56
9.6 GDB features for C++	56
10. Examining the Symbol Table	58
11. Altering Execution	61
11.1 Assignment to variables	61
11.2 Continuing at a different address	61
11.3 Giving your program a signal	62
11.4 Returning from a function	62
11.5 Calling program functions	63
11.6 Patching programs	63
12. GDB Files	64
12.1 Commands to specify files	64
12.2 Errors reading symbol files	67
13. Specifying a Debugging Target	69
13.1 Active targets	69
13.2 Commands for managing targets	69
13.3 Choosing target byte order	70
13.4 Remote debugging	70
14. Controlling GDB	71
14.1 Prompt	71
14.2 Command editing	71
14.3 Command history	71
14.4 Screen size	72
14.5 Numbers	73
14.6 Optional warnings and messages	73
15. Canned Sequences of Commands	75
15.1 User-defined commands	75
15.2 User-defined command hooks	76
15.3 Command files	76
15.4 Commands for controlled output	77
16. Renamed Commands	78

1. A Sample GDB Session

You can use this manual at your leisure to read all about GDB. However, a handful of commands are enough to get started using the debugger. This chapter illustrates those commands.

In this sample session, we emphasize user input like this: **input**, to make it easier to pick out from the surrounding output.

One of the preliminary versions of GNU m4 (a generic macro processor) exhibits the following bug: sometimes, when we change its quote strings from the default, the commands used to capture one macro definition within another stop working. In the following short m4 session, we define a macro `foo` which expands to `0000`; we then use the m4 built-in `defn` to define `bar` as the same thing. However, when we change the open quote string to `<QUOTE>` and the close quote string to `<UNQUOTE>`, the same procedure fails to define a new synonym `baz`:

```
$ cd gnu/m4
$ ./m4
define(foo,0000)

foo
0000
define(bar,defn('foo'))

bar
0000
changequote(<QUOTE>,<UNQUOTE>)

define(baz,defn(<QUOTE>foo<UNQUOTE>))
baz
C-d
m4: End of input: 0: fatal error: EOF in string
```

Let us use GDB to try to see what is going on.

```
$ gdb m4}
GDB is free software and you are welcome to distribute copies
of it under certain conditions; type "show copying" to see
the conditions.
There is absolutely no warranty for GDB; type "show warranty"
for details.

GDB 4.16.1, Copyright 1995 Free Software Foundation, Inc...
(gdb)
```

GDB reads only enough symbol data to know where to find the rest when needed; as a result, the first prompt comes up very quickly. We now tell GDB to use a narrower display width than usual, so that examples fit in this manual.

```
(gdb) set width 70
```

We need to see how the m4 built-in `changequote` works. Having looked at the source, we know the relevant subroutine is `m4_changequote`, so we set a breakpoint there with the GDB `break` command.

```
(gdb) break m4_changequote
Breakpoint 1 at 0x62f4: file builtin.c, line 879.
```

Using the `run` command, we start m4 running under GDB control; as long as control does not reach the `m4_changequote` subroutine, the program runs as usual:

```
(gdb) run
Starting program: /work/Editorial/gdb/gnu/m4/m4
define(foo,0000)

foo
```

0000

To trigger the breakpoint, we call `changequote`. GDB suspends execution of `m4`, displaying information about the context where it stops.

changequote(<QUOTE>, <UNQUOTE>)

```
Breakpoint 1, m4_changequote (argc=3, argv=0x33c70)
  at builtin.c:879
879      if (bad_argc(TOKEN_DATA_TEXT(argv[0]), argc, 1, 3))
```

Now we use the command `n` (`next`) to advance execution to the next line of the current function.

```
(gdb) n
882      set_quotes((argc >= 2) ? TOKEN_DATA_TEXT(argv[1]) : nil,
```

`set_quotes` looks like a promising subroutine. We can go into it by using the command `s` (`step`) instead of `next`. `step` goes to the next line to be executed in **any** subroutine, so it steps into `set_quotes`.

```
(gdb) s
set_quotes (lq=0x34c78 "<QUOTE>", rq=0x34c88 "<UNQUOTE>")
  at input.c:530
530      if (lquote != def_lquote)
```

The display that shows the subroutine where `m4` is now suspended (and its arguments) is called a stack frame display. It shows a summary of the stack. We can use the `backtrace` command (which can also be spelled `bt`), to see where we are in the stack as a whole: the `backtrace` command displays a stack frame for each active subroutine.

```
(gdb) bt
#0  set_quotes (lq=0x34c78 "<QUOTE>", rq=0x34c88 "<UNQUOTE>")
    at input.c:530
#1  0x6344 in m4_changequote (argc=3, argv=0x33c70)
    at builtin.c:882
#2  0x8174 in expand_macro (sym=0x33320) at macro.c:242
#3  0x7a88 in expand_token (obs=0x0, t=209696, td=0xf7fffa30)
    at macro.c:71
#4  0x79dc in expand_input () at macro.c:40
#5  0x2930 in main (argc=0, argv=0xf7fffb20) at m4.c:195
```

We step through a few more lines to see what happens. The first two times, we can use `s`; the next two times we use `n` to avoid falling into the `xstrdup` subroutine.

```
(gdb) s
0x3b5c 532      if (rquote != def_rquote)
(gdb) s
0x3b80 535      lquote = (lq == nil || *lq == ' ') ? def_lquote : xstrdup(lq);
(gdb) n
536      rquote = (rq == nil || *rq == ' ') ? def_rquote : xstrdup(rq);
(gdb) n
538      len_lquote = strlen(rquote);
```

The last line displayed looks a little odd; we can examine the variables `lquote` and `rquote` to see if they are in fact the new left and right quotes we specified. We use the command `p` (`print`) to see their values.

```
(gdb) p lquote
$1 = 0x35d40 "<QUOTE>"
(gdb) p rquote
$2 = 0x35d50 "<UNQUOTE>"
```

`lquote` and `rquote` are indeed the new left and right quotes. To look at some context, we can display ten lines of source surrounding the current line with the `l` (`list`) command.

```

(gdb) l
533         xfree(rquote);
534
535         lquote = (lq == nil || *lq == ' ') ? def_lquote : xstrdup (lq);
536         rquote = (rq == nil || *rq == ' ') ? def_rquote : xstrdup (rq);
537
538         len_lquote = strlen(rquote);
539         len_rquote = strlen(lquote);
540     }
541
542     void

```

Let us step past the two lines that set `len_lquote` and `len_rquote`, and then examine the values of those variables.

```

(gdb) n
539         len_rquote = strlen(lquote);
(gdb) n
540     }
(gdb) p len_lquote
$3 = 9
(gdb) p len_rquote
$4 = 7

```

That certainly looks wrong, assuming `len_lquote` and `len_rquote` are meant to be the lengths of `lquote` and `rquote` respectively. We can set them to better values using the `p` command, since it can print the value of any expression—and that expression can include subroutine calls and assignments.

```

(gdb) p len_lquote=strlen(lquote)
$5 = 7
(gdb) p len_rquote=strlen(rquote)
$6 = 9

```

Is that enough to fix the problem of using the new quotes with the `m4` built-in `defn`? We can allow `m4` to continue executing with the `c` (`continue`) command, and then try the example that caused trouble initially:

```

(gdb) c
Continuing.

define(baz,defn(<QUOTE>foo<UNQUOTE>))

baz
0000

```

Success! The new quotes now work just as well as the default ones. The problem seems to have been just the two typos defining the wrong lengths. We allow `m4` exit by giving it an EOF as input:

```

C-d
Program exited normally.

```

The message **Program exited normally.** is from GDB; it indicates `m4` has finished executing. We can end our GDB session with the GDB `quit` command.

```

(gdb) quit

```

2. Getting In and Out of GDB

This chapter discusses how to start GDB, and how to get out of it. The essentials are:

- type **gdb** to start GDB.
- type **quit** or **C-d** to exit.

2.1 Invoking GDB

Invoke GDB by running the program **gdb**. Once started, GDB reads commands from the terminal until you tell it to exit.

You can also run **gdb** with a variety of arguments and options, to specify more of your debugging environment at the outset.

The command-line options described here are designed to cover a variety of situations; in some environments, some of these options may effectively be unavailable.

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

```
gdb program
```

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

```
gdb program core
```

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

```
gdb program 1234
```

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

Taking advantage of the second command-line argument requires a fairly complete operating system; when you use GDB as a remote debugger attached to a bare board, there may not be any notion of “process”, and there is often no way to get a core dump.

You can run **gdb** without printing the front material, which describes GDB’s non-warranty, by specifying **-silent**:

```
gdb -silent
```

You can further control how GDB starts up by using command-line options. GDB itself can remind you of the options available.

Type

```
gdb -help
```

to display all available options and briefly describe their use (**gdb -h** is a shorter equivalent).

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

2.1.1 Choosing files

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

Many options have both long and short forms; both are shown in the following list. GDB also recognizes the long forms if you truncate them, so long as enough of the option is present to be unambiguous. (If you prefer, you can flag option arguments with **--** rather than **-**, though we

illustrate the more usual convention.)

-symbols file

-s file Read symbol table from file *file*.

-exec file

-e file Use file *file* as the executable file to execute when appropriate.

-se file

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

-core file

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

-c number

Connect to process ID *number*, as with the `attach` command (unless there is a file in core-dump format named *number*, in which case `-c` specifies that file as a core dump to read).

-command file

-x file Execute GDB commands from file *file*. See *Command files*.

-directory directory

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

-mapped

-m

Warning: this option depends on operating system facilities that are not supported on all systems. If memory-mapped files are available on your system through the `mmap` system call, you can use this option to have GDB write the symbols from your program into a reusable file in the current directory. If the program you are debugging is called `/tmp/fred`, the mapped symbol file is `./fred.syms`. Future GDB debugging sessions notice the presence of this file, and can quickly map in symbol information from it, rather than reading the symbol table from the executable program.

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

-readnow

-r

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

The `-mapped` and `-readnow` options are typically combined in order to build a `.syms` file that contains complete symbol information. (See *Commands to specify files*, for information.)

A `.syms` file for future use is:

```
gdb -batch -nx -mapped -readnow programname
```

2.1.2 Choosing modes

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

-nx

-n

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

- quiet**
-q “Quiet”. Do not print the introductory and copyright messages. These messages are also suppressed in batch mode.
- batch** Run in batch mode. Exit with status 0 after processing all the command files specified with **-x** (and all commands from initialization files, if not inhibited with **-n**). Exit with nonzero status if an error occurs in executing the GDB commands in the command files.
- Batch mode may be useful for running GDB as a filter, for example to download and run a program on another computer; in order to make this more useful, the message
- Program exited normally.
- (which is ordinarily issued whenever a program running under GDB control terminates) is not issued when running in batch mode.
- cd *directory*** Run GDB using *directory* as its working directory, instead of the current directory.
- fullname**
-f GNU Emacs sets this option when it runs GDB as a subprocess. It tells GDB to output the full file name and line number in a standard, recognizable fashion each time a stack frame is displayed (which includes each time your program stops). This recognizable format looks like two 32 characters, followed by the file name, line number and character position separated by colons, and a newline. The Emacs-to-GDB interface program uses the two 32 characters as a signal to display the source code for the frame.
- b *bps*** Set the line speed (baud rate or bits per second) of any serial interface used by GDB for remote debugging.
- tty *device*** Run using *device* for your program’s standard input and output.

2.2 Quitting GDB

- quit** To exit GDB, use the **quit** command (abbreviated **q**), or type an end-of-file character (usually **C-d**). If you do not supply *expression*, GDB will terminate normally; otherwise it will terminate using the result of *expression* as the error code.
- interrupt** An interrupt (often **C-c**) does not exit from GDB, but rather terminates the action of any GDB command that is in progress and returns to GDB command level. It is safe to type the interrupt character at any time because GDB does not allow it to take effect until a time when it is safe.
- If you have been using GDB to control an attached process or device, you can release it with the **detach** command.

2.3 Shell commands

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

shell *command string*

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

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

make *make-args*

Execute the **make** program with the specified arguments. This is equivalent to **shell make** *make-args*.

3. GDB Commands

You can abbreviate a GDB command to the first few letters of the command name, if that abbreviation is unambiguous; and you can repeat certain GDB commands by typing just RET. You can also use the TAB key to get GDB to fill out the rest of a word in a command (or to show you the alternatives available, if there is more than one possibility).

3.1 Command syntax

A GDB command is a single line of input. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. For example, the command `step` accepts an argument which is the number of times to step, as in `step 5`. You can also use the `step` command with no arguments. Some command names do not allow any arguments.

GDB command names may always be truncated if that abbreviation is unambiguous. Other possible command abbreviations are listed in the documentation for individual commands. In some cases, even ambiguous abbreviations are allowed; for example, `s` is specially defined as equivalent to `step` even though there are other commands whose names start with `s`. You can test abbreviations by using them as arguments to the `help` command.

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

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

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

Any text from a `#` to the end of the line is a comment; it does nothing. This is useful mainly in command files.

3.2 Command completion

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

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

```
(gdb) info bre TAB
```

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

```
(gdb) info breakpoints
```

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

If there is more than one possibility for the next word when you press TAB, GDB sounds a bell. You can either supply more characters and try again, or just press TAB a second time; GDB displays all the possible completions for that word. For example, you might want to set a breakpoint on a subroutine whose name begins with **make_**, but when you type **b make_TAB** GDB just sounds the bell. Typing TAB again displays all the function names in your program

that begin with those characters, for example:

```
(gdb) b make_ TAB
GDB sounds bell; press TAB again, to see:
make_a_section_from_file      make_envirom
make_abs_section              make_function_type
make_blockvector              make_pointer_type
make_cleanup                  make_reference_type
make_command                  make_symbol_completion_list
(gdb) b make_
```

After displaying the available possibilities, GDB copies your partial input (**b make_** in the example) so you can finish the command.

If you just want to see the list of alternatives in the first place, you can press **M-?** rather than pressing TAB twice. **M-?** means **META ?**. You can type this either by holding down a key designated as the META shift on your keyboard (if there is one) while typing **?**, or as ESC followed by **?**.

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

The most likely situation where you might need this is in typing the name of a C++ function. This is because C++ allows function overloading (multiple definitions of the same function, distinguished by argument type). For example, when you want to set a breakpoint you may need to distinguish whether you mean the version of **name** that takes an **int** parameter, **name(int)**, or the version that takes a **float** parameter, **name(float)**. To use the word-completion facilities in this situation, type a single quote **'** at the beginning of the function name. This alerts GDB that it may need to consider more information than usual when you press TAB or **M-?** to request word completion:

```
(gdb) b 'bubble( M-?
bubble(double,double)      bubble(int,int)
(gdb) b 'bubble(
```

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

```
(gdb) b bub TAB
GDB alters your input line to the following, and rings a bell:
(gdb) b 'bubble(
```

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

3.3 Getting help

You can always ask GDB itself for information on its commands, using the command **he lp**.

help

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

```
(gdb) help
List of classes of commands:
running -- Running the program
stack -- Examining the stack
data -- Examining data
breakpoints -- Making program stop at certain points
files -- Specifying and examining files
```

```

status -- Status inquiries
support -- Support facilities
user-defined -- User-defined commands
aliases -- Aliases of other commands
obscure -- Obscure features

Type "help" followed by a class name for a list of
commands in that class.
Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)

```

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

```

(gdb) help status
Status inquiries.

List of commands:

show -- Generic command for showing things set
      with "set"
info -- Generic command for printing status

Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)

```

help command

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

complete args

The `complete args` command lists all the possible completions for the beginning of a command. Use `args` to specify the beginning of the command you want completed. For example:

```

complete i
info
inspect
ignore

```

In addition to `help`, you can use the GDB commands `info` and `show` to inquire about the state of your program, or the state of GDB itself. Each command supports many topics of inquiry; this manual introduces each of them in the appropriate context. The listings under `info` and under `show` in the Index point to all the sub-commands.

info This command (abbreviated `i`) is for describing the state of your program. For example, you can list the arguments given to your program with `info args`, list the registers currently in use with `info registers`, or list the breakpoints you have set with `info breakpoints`. You can get a complete list of the `info` sub-commands with `help info`.

set You can assign the result of an expression to an environment variable with `set`. For example, you can set the GDB prompt to a `$`-sign with `set prompt $`.

show In contrast to `info`, `show` is for describing the state of GDB itself. You can change most of the things you can `show`, by using the related command `set`; for example, you can control what number system is used for displays with `set`

`radix`, or simply inquire which is currently in use with `show radix`.

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

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

show version

Show what version of GDB is running. You should include this information in GDB bug-reports. If multiple versions of GDB are in use at your site, you may occasionally want to determine which version of GDB you are running; as GDB evolves, new commands are introduced, and old ones may wither away. The version number is also announced when you start GDB.

show copying

Display information about permission for copying GDB.

show warranty

Display the GNU “NO WARRANTY” statement.

4. Running Programs Under GDB

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

4.1 Compiling for debugging

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

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

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

GCC, the GNU C compiler, supports **-g** with or without **-O**, making it possible to debug optimized code. We recommend that you **always** use **-g** whenever you compile a program. You may think your program is correct, but there is no sense in pushing your luck.

When you debug a program compiled with **-g -O**, remember that the optimizer is rearranging your code; the debugger shows you what is really there. Do not be too surprised when the execution path does not exactly match your source file! An extreme example: if you define a variable, but never use it, GDB never sees that variable—because the compiler optimizes it out of existence.

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

Older versions of the GNU C compiler permitted a variant option **-gg** for debugging information. GDB no longer supports this format; if your GNU C compiler has this option, do not use it.

4.2 Starting your program

run

r Use the **run** command to start your program under GDB. You must first specify the program name with an argument to GDB, or by using the **file** or **exec-file** command.

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

The execution of a program is affected by certain information it receives from its superior. GDB provides ways to specify this information, which you must do **before** starting your program. (You can change it after starting your program, but such changes only affect your program the next time you start it.) This information may be divided into four categories:

The arguments.

Specify the arguments to give your program as the arguments of the **run** command. If a shell is available on your target, the shell is used to pass the arguments, so that you may use normal conventions (such as wildcard expansion or variable substitution) in describing the arguments. In Unix systems, you can control which shell is used with the **SHELL** environment variable.

The environment.

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

The working directory.

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

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. *Your program's input and output.*

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

When you issue the `run` command, your program begins to execute immediately. *Stopping and continuing*, for discussion of how to arrange for your program to stop. Once your program has stopped, you may call functions in your program, using the `print` or `call` commands. *Examining Data.*

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

4.3 Your program's arguments

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

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

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

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

4.4 Your program's environment

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

path directory

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

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

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

show environment [*varname*]

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

set environment *varname* [=] *value*

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

For example, this command:

```
set env USER = foo
```

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

unset environment *varname*

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

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

4.5 Your program's working directory

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

cd *directory* Set the GDB working directory to *directory*.

pwd Print the GDB working directory.

4.6 Your program's input and output

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

info terminal

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

You can redirect your program's input and/or output using shell redirection with the `run` command. For example,

```
run > outfile
```

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

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

```
tty /dev/ttyb
```

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

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

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

4.7 Debugging an already-running process

attach *process-id*

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

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

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

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

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

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

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

4.8 Killing the child process

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

This command is useful if you wish to debug a core dump instead of a running process. GDB

ignores any core dump file while your program is running.

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

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

4.9 Additional process information

Some operating systems provide a facility called `/proc` that can be used to examine the image of a running process using file-system subroutines. If GDB is configured for an operating system with this facility, the command `info proc` is available to report on several kinds of information about the process running your program. `info proc` works only on SVR4 systems that support `procfs`.

info proc Summarize available information about the process.

info proc mappings

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

info proc times

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

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

info proc status

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

info proc all Show all the above information about the process.

4.10 Debugging programs with multiple threads

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

GDB provides these facilities for debugging multi-thread programs:

- automatic notification of new threads
- **thread** *threadno*, a command to switch among threads
- **info threads**, a command to inquire about existing threads
- **thread apply** [*threadno*] [**all**] *args*
a command to apply a command to a list of threads
- **thread-specific breakpoints**

Warning: These facilities are not yet available on every GDB configuration where the operating system supports threads. If your GDB does not support threads, these commands have no effect. For example, a system without thread support shows no output from **info threads**, and always rejects the `thread` command, like this:

```
(gdb) info threads
(gdb) thread 1
Thread ID 1 not known. Use the "info threads" command to
see the IDs of currently known threads.
```

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

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

```
[New process 35 thread 27]
```

when GDB notices a new thread. In contrast, on an SGI system, the *systag* is simply something like **process 368**, with no further qualifier.

For debugging purposes, GDB associates its own thread number—always a single integer—with each thread in your program.

info threads Display a summary of all threads currently in your program. GDB displays for each thread (in this order):

1. the thread number assigned by GDB
2. the target system's thread identifier (*systag*)
3. the current stack frame summary for that thread

An asterisk ***** to the left of the GDB thread number indicates the current thread.

For example,

```
(gdb) info threads
3 process 35 thread 27 0x34e5 in sigpause ()
2 process 35 thread 23 0x34e5 in sigpause ()
* 1 process 35 thread 13 main (argc=1, argv=0x7ffffff8)
  at threadtest.c:68
```

thread threadno

Make thread number *threadno* the current thread. The command argument *threadno* is the internal GDB thread number, as shown in the first field of the **info threads** display. GDB responds by displaying the system identifier of the thread you selected, and its current stack frame summary:

```
(gdb) thread 2
[Switching to process 35 thread 23]
0x34e5 in sigpause ()
```

As with the **[New ...]** message, the form of the text after **Switching to** depends on your system's conventions for identifying threads.

thread apply [threadno] [all] args

The **thread apply** command allows you to apply a command to one or more threads. Specify the numbers of the threads that you want affected with the command argument *threadno*. *threadno* is the internal GDB thread number, as shown in the first field of the **info threads** display. To apply a command to all threads, use **thread apply all args**.

Whenever GDB stops your program, due to a breakpoint or a signal, it automatically selects the thread where that breakpoint or signal happened. GDB alerts you to the context switch with a message of the form **[Switching to systag]** to identify the thread.

See *Stopping and starting multi-thread programs*, for more information about how GDB behaves when you stop and start programs with multiple threads.

See *Setting watchpoints*, for information about watchpoints in programs with multiple threads.

4.11 Debugging programs with multiple processes

GDB has no special support for debugging programs which create additional processes using the `fork` function. When a program forks, GDB will continue to debug the parent process and the child process will run unimpeded. If you have set a breakpoint in any code which the child then executes, the child will get a `SIGTRAP` signal which (unless it catches the signal) will cause it to terminate.

However, if you want to debug the child process there is a workaround which isn't too painful. Put a call to `sleep` in the code which the child process executes after the fork. It may be useful to sleep only if a certain environment variable is set, or a certain file exists, so that the delay need not occur when you don't want to run GDB on the child. While the child is sleeping, use the `ps` program to get its process ID. Then tell GDB (a new invocation of GDB if you are also debugging the parent process) to attach to the child process (see *Attach*). From that point on you can debug the child process just like any other process which you attached to.

5. Stopping and Continuing

The principal purposes of using a debugger are so that you can stop your program before it terminates; or so that, if your program runs into trouble, you can investigate and find out why.

Inside GDB, your program may stop for any of several reasons, such as a signal, a breakpoint, or reaching a new line after a GDB command such as `step`. You may then examine and change variables, set new breakpoints or remove old ones, and then continue execution. Usually, the messages shown by GDB provide ample explanation of the status of your program—but you can also explicitly request this information at any time.

info program

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

5.1 Breakpoints, watchpoints, and exceptions

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

In SunOS 4.x, SVR4, and Alpha OSF/1 configurations, you can now set breakpoints in shared libraries before the executable is run.

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

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

5.1.1 Setting breakpoints

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

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

break function

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

break +offset

break -offset

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

break *linenum*

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

break *filename:linenum*

Set a breakpoint at line *linenum* in source file *filename*.

break *filename:function*

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

break **address*

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

break

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

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

break ... if *cond*

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

tbreak *args*

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

hbreak *args*

Set a hardware-assisted breakpoint. *args* are the same as for the **break** command and the breakpoint is set in the same way, but the breakpoint requires hardware support and some target hardware may not have this support. The main purpose of this is EPROM/ROM code debugging, so you can set a breakpoint at an instruction without changing the instruction. This can be used with the new trap-generation provided by SPARClike DSU. DSU will generate traps when a program accesses some data or instruction address that is assigned to the debug registers. However the hardware breakpoint registers can only take two data breakpoints, and GDB will reject this command if more than two are used. Delete or disable unused hardware breakpoints before setting new ones. See *Break conditions*.

thbreak *args*

Set a hardware-assisted breakpoint enabled only for one stop. *args* are the same as for the **hbreak** command and the breakpoint is set in the same way. However, like the **tbreak** command, the breakpoint is automatically deleted after the first time your

program stops there. Also, like the **hbreak** command, the breakpoint requires hardware support and some target hardware may not have this support. *Disabling breakpoints.* Also see *Break conditions.*

rbreak *regex*

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

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

info breakpoints [*n*]

info break [*n*]

info watchpoints [*n*]

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

Breakpoint Numbers

Type Breakpoint or watchpoint.

Disposition

Whether the breakpoint is marked to be disabled or deleted when hit.

Enabled or Disabled

Enabled breakpoints are marked with **y**. **n** marks breakpoints that are not enabled.

Address

Where the breakpoint is in your program, as a memory address

What

Where the breakpoint is in the source for your program, as a file and line number.

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

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

info break now displays a count of the number of times the breakpoint has been hit. This is especially useful in conjunction with the **ignore** command. You can ignore a large number of breakpoint hits, look at the breakpoint info to see how many times the breakpoint was hit, and then run again, ignoring one less than that number. This will get you quickly to the last hit of that breakpoint.

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

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

You can see these breakpoints with the GDB maintenance command **maint info breakpoints**.

maint info breakpoints

Using the same format as **info breakpoints**, display both the breakpoints you've

set explicitly, and those GDB is using for internal purposes. Internal breakpoints are shown with negative breakpoint numbers. The type column identifies what kind of breakpoint is shown:

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

5.1.2 Setting watchpoints

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

Watchpoints currently execute two orders of magnitude more slowly than 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 *expr* Set a watchpoint for an expression. GDB will break when *expr* is written into by the program and its value changes. This can be used with the new trap-generation provided by SPARClite DSU. DSU will generate traps when a program accesses some data or instruction address that is assigned to the debug registers. For the data addresses, DSU facilitates the `watch` command. However the hardware breakpoint registers can only take two data watchpoints, and both watchpoints must be the same kind. For example, you can set two watchpoints with `watch` commands, two with `rwatch` commands, or two with `awatch` commands, but you cannot set one watchpoint with one command and the other with a different command. GDB will reject the command if you try to mix watchpoints. Delete or disable unused watchpoint commands before setting new ones.

rwatch *expr* Set a watchpoint that will break when *args* is read by the program. If you use both watchpoints, both must be set with the `rwatch` command.

awatch *expr* Set a watchpoint that will break when *args* is read and written into by the program. If you use both watchpoints, both must be set with the `awatch` command.

info watchpoints

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

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

5.1.3 Breakpoints and exceptions

Some languages, such as GNU C++, implement exception handling. You can use GDB to examine what caused your program to raise an exception, and to list the exceptions your program is prepared to handle at a given point in time.

catch exceptions

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

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

There are currently some limitations to exception handling in GDB:

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

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

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

```
/* addr is where the exception identifier is stored.
   ID is the exception identifier. */
void __raise_exception (void **addr, void *id);
```

To make the debugger catch all exceptions before any stack unwinding takes place, set a breakpoint on `__raise_exception()`.

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

5.1.4 Deleting breakpoints

It is often necessary to eliminate a breakpoint or watchpoint once it has done its job and you no longer want your program to stop there. This is called *deleting* the breakpoint. A breakpoint that has been deleted no longer exists; it is forgotten.

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

It is not necessary to delete a breakpoint to proceed past it. GDB automatically ignores breakpoints on the first instruction to be executed when you continue execution without changing the execution address.

clear

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

clear function**clear filename:function**

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

clear linenum**clear filename:linenum**

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

delete [breakpoints] [bnums...]

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

5.1.5 Disabling breakpoints

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

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

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

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

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

disable [breakpoints] [bnums...]

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

enable [breakpoints] [bnums...]

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

enable [breakpoints] once bnums...

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

enable [breakpoints] delete bnums...

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

Except for a breakpoint set with `tbreak`, breakpoints that you set are initially enabled; subsequently, they become disabled or enabled only when you use one of the commands above. (The command `until` can set and delete a breakpoint of its own, but it does not change the state of your other breakpoints; see *Continuing and stepping*.)

5.1.6 Break conditions

The simplest sort of breakpoint breaks every time your program reaches a specified place. You can also specify a *condition* for a breakpoint. A condition is just a Boolean expression in your programming language. A breakpoint with a condition evaluates the expression each time your program reaches it, and your 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.

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

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

Break conditions can be specified when a breakpoint is set, by using **if** in the arguments to the **break** command. They can also be changed at any time with the **condition** command. The **watch** command does not recognize the **if** keyword; **condition** is the only way to impose a further condition on a watchpoint.

condition *bnum expression*

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

condition *bnum*

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

A special case of a breakpoint condition is to stop only when the breakpoint has been reached a certain number of times. This is so useful that there is a special way to do it, using the **ignore count** of the breakpoint. Every breakpoint has an ignore count, which is an integer. Most of the time, the ignore count is zero, and therefore has no effect. But if your program reaches a breakpoint whose ignore count is positive, then instead of stopping, it just decrements the ignore count by one and continues. As a result, if the ignore count value is *n*, the breakpoint does not stop the next *n* times your program reaches it.

ignore *bnum count*

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

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

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

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

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

5.1.7 Breakpoint command lists

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

```
commands [bnum]  
... command-list ...  
end
```

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

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

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

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

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

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

If the first command you specify in a command list is **silent**, the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If none of the remaining commands print anything, you see no sign that the breakpoint was reached. **silent** is meaningful only at the beginning of a breakpoint command list.

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

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

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

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

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

5.1.8 Breakpoint menus

Some programming languages (notably C++) permit a single function name to be defined several times, for application in different contexts. This is called *overloading*. When a function name is overloaded, **break function** is not enough to tell GDB where you want a breakpoint. If you realize this is a problem, you can use something like **break function(types)** to specify which particular version of the function you want. Otherwise, GDB offers you a menu of numbered choices for different possible breakpoints, and waits for your selection with the prompt **>**. The first two options are always **[0] cancel** and **[1] all**. Typing **1** sets a breakpoint at each definition of *function*, and typing **0** aborts the **break** command without setting any new breakpoints.

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

```
(gdb) b String::after
[0] cancel
[1] all
[2] file:String.cc; line number:867
[3] file:String.cc; line number:860
[4] file:String.cc; line number:875
[5] file:String.cc; line number:853
[6] file:String.cc; line number:846
[7] file:String.cc; line number:735
> 2 4 6
Breakpoint 1 at 0xb26c: file String.cc, line 867.
Breakpoint 2 at 0xb344: file String.cc, line 875.
Breakpoint 3 at 0xafcc: file String.cc, line 846.
Multiple breakpoints were set.
Use the "delete" command to delete unwanted
breakpoints.
(gdb)
```

5.2 Continuing and stepping

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

continue [*ignore-count*]

c [*ignore-count*]

fg [*ignore-count*]

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

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

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

To resume execution at a different place, you can use **return** to go back to the calling function; or **jump** to go to an arbitrary location in your program.

A typical technique for using stepping is to set a breakpoint at the beginning of the function or the section of your program where a problem is believed to lie, run your program until it stops at that breakpoint, and then step through the suspect area, examining the variables that are interesting,

until you see the problem happen.

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

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

The `step` command now only stops at the first instruction of a source line. This prevents the multiple stops that used to occur in switch statements, for loops, etc. `step` continues to stop if a function that has debugging information is called within the line.

Also, the `step` command now only enters a subroutine if there is line number information for the subroutine. Otherwise it acts like the `next` command. This avoids problems when using `cc -gl` on MIPS machines. Previously, `step` entered subroutines if there was any debugging information about the routine.

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

next [count] Continue to the next source line in the current (innermost) stack frame. This is similar to `step`, but function calls that appear within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the original stack level that was executing when you gave the `next` command. This command is abbreviated `n`.

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

The `next` command now only stops at the first instruction of a source line. This prevents the multiple stops that used to occur in switch statements, for loops, etc.

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

Contrast this with the `return` command.

**u
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, which forces you to step through the next iteration.

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

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

```
(gdb) f
```

```
#0  main (argc=4, argv=0xf7fffae8) at m4.c:206
206      expand_input();
(gdb) until
195      for ( ; argc > 0; NEXTARG) {
```

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

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

until *location*

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

stepi

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

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

An argument is a repeat count, as in `step`.

nexti

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

An argument is a repeat count, as in `next`.

5.3 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, in Unix `SIGINT` is the signal a program gets when you type an interrupt (often `C-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 your program has requested an alarm).

Some signals, including `SIGALRM`, are a normal part of the functioning of your program. Others, such as `SIGSEGV`, indicate errors; these signals are *fatal* (kill your program immediately) if the program has not specified in advance some other way to handle the signal. `SIGINT` does not indicate an error in your program, but it is normally fatal so it can carry out the purpose of the interrupt: to kill the program.

GDB has the ability to detect any occurrence of a signal in your program. 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 your program) but to stop your program immediately whenever an error signal happens. You can change these settings with the `handle` command.

info signals 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.

`info handle` is the new alias for `info signals`.

handle *signal keywords...*

Change the way GDB handles signal *signal*. *signal* can be the number of a signal or its name (with or without the **SIG** at the beginning). The *keywords* say what change to make.

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

nostop	GDB should not stop your program when this signal happens. It may still print a message telling you that the signal has come in.
stop	GDB should stop your program when this signal happens. This implies the print keyword as well.
print	GDB should print a message when this signal happens.
noprint	GDB should not mention the occurrence of the signal at all. This implies the nostop keyword as well.
pass	GDB should allow your program to see this signal; your program can handle the signal, or else it may terminate if the signal is fatal and not handled.
nopass	GDB should not allow your program to see this signal.

When a signal stops your program, the signal is not visible until you continue. Your program sees 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 your program sees that signal when you continue.

You can also use the **signal** command to prevent your program from seeing a signal, or cause it to see a signal it normally would not see, or to give it any signal at any time. For example, if your 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 your program would probably terminate immediately as a result of the fatal signal once it saw the signal. To prevent this, you can continue with **signal 0**.

5.4 Stopping and starting multi-thread programs

When your program has multiple threads, you can choose whether to set breakpoints on all threads, or on a particular thread.

break linespec thread threadno**break linespec thread threadno if ...**

linespec specifies source lines; there are several ways of writing them, but the effect is always to specify some source line.

Use the qualifier **thread threadno** with a breakpoint command to specify that you only want GDB to stop the program when a particular thread reaches this breakpoint. *threadno* is one of the numeric thread identifiers assigned by GDB, shown in the first column of the **info threads** display.

If you do not specify **thread threadno** when you set a breakpoint, the breakpoint applies to **all** threads of your program.

You can use the **thread** qualifier on conditional breakpoints as well; in this case, place **thread threadno** before the breakpoint condition, like this:

```
(gdb) break frik.c:13 thread 28 if bartab > lim
```

Whenever your program stops under GDB for any reason, **all** threads of execution stop, not just the current thread. This allows you to examine the overall state of the program, including switching between threads, without worrying that things may change underfoot.

Conversely, whenever you restart the program, **all** threads start executing. **This is true even**

when single-stepping with commands like `step` or `next`.

In particular, GDB cannot single-step all threads in lockstep. Since thread scheduling is up to your debugging target's operating system (not controlled by GDB), other threads may execute more than one statement while the current thread completes a single step. Moreover, in general other threads stop in the middle of a statement, rather than at a clean statement boundary, when the program stops.

You might even find your program stopped in another thread after continuing or even single-stepping. This happens whenever some other thread runs into a breakpoint, a signal, or an exception before the first thread completes whatever you requested.

6. 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, information about the call is generated. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a *stack frame*. 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 of this information.

One of the stack frames is *selected* by GDB and many GDB commands refer implicitly to the selected frame. In particular, whenever you ask GDB for the value of a variable in your program, the value is found in the selected frame. There are special GDB commands to select whichever frame you are interested in. *Selecting a frame*.

When your program stops, GDB automatically selects the currently executing frame and describes it briefly, similar to the **frame** command.

6.1 Stack frames

The call stack is divided up into contiguous pieces called **stack frames**, or *frames* for short; 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 one byte whose address serves 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 zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your program; they are assigned by GDB to give you a way of designating stack frames in GDB commands.

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.

frame [*args*] The **frame** command allows you to move from one stack frame to another, and to print the stack frame you select. *args* may be either the address of the frame or the stack frame number. Without an argument, **frame** prints the current stack frame.

select-frame

The **select-frame** command allows you to move from one stack frame to another without printing the frame. This is the silent version of **frame**.

6.2 Backtraces

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

backtrace

bt 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 **C-c**.

backtrace n

bt n Similar, but print only the innermost *n* frames.

backtrace -n

bt -n Similar, but print only the outermost *n* frames.

The names `where` and `info stack` (abbreviated `info s`) are additional aliases for `backtrace`.

Each line in the backtrace shows the frame number and the function name. The program counter value is also shown—unless you use `set print address off`. The backtrace also shows the source file name and line number, as well as the arguments to the function. The program counter value is omitted if it is at the beginning of the code for that line number.

Here is an example of a backtrace. It was made with the command **bt 3**, so it shows the innermost three frames.

```
#0  m4_traceon (obs=0x24eb0, argc=1, argv=0x2b8c8)
    at builtin.c:993
#1  0x6e38 in expand_macro (sym=0x2b600) at macro.c:242
#2  0x6840 in expand_token (obs=0x0, t=177664, td=0xf7fffb08)
    at macro.c:71
(More stack frames follow...)
```

The display for frame zero does not begin with a program counter value, indicating that your program has stopped at the beginning of the code for line 993 of `builtin.c`.

6.3 Selecting a frame

Most commands for examining the stack and other data in your program work on whichever stack frame is selected at the moment. Here are the commands for selecting a stack frame; all of them finish by printing a brief description of the stack frame just selected.

frame n

f n Select frame number *n*. Recall that frame zero is the innermost (currently executing) frame, frame one is the frame that called the innermost one, and so on. The highest-numbered frame is the one for `main`.

frame addr

f addr Select 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 when your program has multiple stacks and switches between them.

On the SPARC architecture, `frame` needs two addresses to select an arbitrary frame: a frame pointer and a stack pointer.

On the MIPS and Alpha architecture, it needs two addresses: a stack pointer and a program counter.

On the 29k architecture, it needs three addresses: a register stack pointer, a

program counter, and a memory stack pointer.

up *n* Move *n* frames up the stack. For positive numbers *n*, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. *n* defaults to one.

down *n* Move *n* frames down the stack. For positive numbers *n*, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. *n* defaults to one. You may abbreviate **down** as **do**.

All of these commands end by printing two lines of output describing the frame. The first line shows the frame number, the function name, the arguments, and the source file and line number of execution in that frame. The second line shows the text of that source line.

For example:

```
(gdb) up
#1 0x22f0 in main (argc=1, argv=0xf7ffbf4, env=0xf7ffbf4)
    at env.c:10
10      read_input_file (argv[i]);
```

After such a printout, the **list** command with no arguments prints ten lines centered on the point of execution in the frame. *Printing source lines.*

up-silently *n*

down-silently *n*

These two commands are variants of **up** and **down**, respectively; they differ in that they do their work silently, without causing display of the new frame. They are intended primarily for use in GDB command scripts, where the output might be unnecessary and distracting.

6.4 Information about a frame

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

frame

f

When used without any argument, this command does not change which frame is selected, but prints a brief description of the currently selected stack frame. It can be abbreviated **f**. With an argument, this command is used to select a stack frame. *Selecting a frame.*

info frame

info f

This command prints a verbose description of the selected stack frame, including:

- the address of the frame
- the address of the next frame down (called by this frame)
- the address of the next frame up (caller of this frame)
- the language in which the source code corresponding to this frame is written
- the address of the frame's arguments
- the program counter saved in it (the address of execution in the caller frame)
- which registers were saved in the frame

something has gone wrong that has made the stack format fail to fit the usual conventions.

info frame *addr*

info f *addr*

Print a verbose description of the frame at address *addr*, without selecting that frame. The selected frame remains unchanged by this command. This requires the same kind of address (more than one for some architectures) that you specify

in the `frame` command. *Selecting a frame.*

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

info locals Print the local variables of the selected frame, each on a separate line. These are all variables (declared either static or automatic) accessible at the point of execution of the selected frame.

info catch Print a list of all the exception handlers that are active in the current stack frame at the current point of execution. To see other exception handlers, visit the associated frame (using the `up`, `down`, or `frame` commands); then type `info catch`.
Breakpoints and exceptions.

7. Examining Source Files

GDB can print parts of your program's source, since the debugging information recorded in the program tells GDB what source files were used to build it. When your program stops, GDB spontaneously prints the line where it stopped. Likewise, when you select a stack frame, GDB prints the line where execution in that frame has stopped. You can print other portions of source files by explicit command.

If you use GDB through its GNU Emacs interface, you may prefer to use Emacs facilities to view source.

7.1 Printing source lines

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

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

- list *linenum*** Print lines centered around line number *linenum* in the current source file.
- list *function*** Print lines centered around the beginning of function *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, this prints lines centered around that line.
- list -** Print lines just before the lines last printed.

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

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

show listsize Display the number of lines that `list` prints.

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

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

- list *linespec*** Print lines centered around the line specified by *linespec*.
- list *first,last*** Print lines from *first* to *last*. Both arguments are linespecs.
- list *,last*** Print lines ending with *last*.
- list *first,*** Print lines starting with *first*.
- list +** Print lines just after the lines last printed.
- list -** Print lines just before the lines last printed.
- list** As described in the preceding table.

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

<i>number</i>	Specifies line <i>number</i> of the current source file. When a list command has two linespecs, this refers to the same source file as the first linespec.
+offset	Specifies the line <i>offset</i> lines after the last line printed. When used as the second linespec in a list command that has two, this specifies the line <i>offset</i> lines down from the first linespec.
-offset	Specifies the line <i>offset</i> lines before the last line printed.
<i>filename:number</i>	Specifies line <i>number</i> in the source file <i>filename</i> .
<i>function</i>	Specifies the line that begins the body of the function <i>function</i> . For example: in C, this is the line with the open brace.
<i>filename:function</i>	Specifies the line of the open-brace that begins the body of the function <i>function</i> in the file <i>filename</i> . You only need the file name with a function name to avoid ambiguity when there are identically named functions in different source files.
*address	Specifies the line containing the program address <i>address</i> . <i>address</i> may be any expression.

7.2 Searching source files

There are two commands for searching through the current source file for a regular expression.

forward-search *regexp*

search *regexp*

The command **forward-search** *regexp* checks each line, starting with the one following the last line listed, for a match for *regexp*. It lists the line that is found. You can use the synonym **search** *regexp* or abbreviate the command name as **fo**.

reverse-search *regexp*

The command **reverse-search** *regexp* 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 is found. You can abbreviate this command as **rev**.

7.3 Specifying source directories

Executable programs sometimes do not record the directories of the source files from which they were compiled, just the names. Even when they do, the directories could be moved between the compilation and your debugging session. GDB has a list of directories to search for source files; this is called the *source path*. Each time GDB wants a source file, it tries all the directories in the list, in the order they are present in the list, until it finds a file with the desired name. Note that the executable search path is **not** used for this purpose. Neither is the current working directory, unless it happens to be in the source path.

If GDB cannot find a source file in the source path, and the object program records a directory, GDB tries that directory too. If the source path is empty, and there is no record of the compilation directory, GDB looks in the current directory as a last resort.

Whenever you reset or rearrange the source path, GDB clears out any information it has cached about where source files are found and where each line is in the file.

When you start GDB, its source path is empty. To add other directories, use the **directory** command.

directory *dirname* ...

dir *dirname* ...

Add directory *dirname* to the front of the source path. Several directory names may be given to this command, separated by **:** or whitespace. You may specify a

directory that is already in the source path; this moves it forward, so GDB searches it sooner.

You can use the string **\$cdir** to refer to the compilation directory (if one is recorded), and **\$cwd** to refer to the current working directory. **\$cwd** is not the same as **.**—the former tracks the current working directory as it changes during your GDB session, while the latter is immediately expanded to the current directory at the time you add an entry to the source path.

directory Reset the source path to empty again. This requires confirmation.

show directories

Print the source path: show which directories it contains.

If your source path is cluttered with directories that are no longer of interest, GDB may sometimes cause confusion by finding the wrong versions of source. You can correct the situation as follows:

1. Use **directory** with no argument to reset the source path to empty.
2. Use **directory** with suitable arguments to reinstall the directories you want in the source path. You can add all the directories in one command.

7.4 Source and machine code

You can use the command **info line** to map source lines to program addresses (and vice versa), and the command **disassemble** to display a range of addresses as machine instructions. When run under GNU Emacs mode, the **info line** command now causes the arrow to point to the line specified. Also, **info line** prints addresses in symbolic form as well as hex.

info line linespec

Print the starting and ending addresses of the compiled code for source line *linespec*. You can specify source lines in any of the ways understood by the **list** command.

For example, we can use **info line** to discover the location of the object code for the first line of function `m4_changequote`:

```
(gdb) info line m4_changecom
Line 895 of "builtin.c" starts at pc 0x634c and ends at 0x6350.
```

We can also inquire (using `*\fPf2addr` as the form for *linespec*) what source line covers a particular address:

```
(gdb) info line *0x63ff
Line 926 of "builtin.c" starts at pc 0x63e4 and ends at 0x6404.
```

After **info line**, 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. Also, this address is saved as the value of the convenience variable **\$_**.

disassemble

This specialized command dumps a range of memory as machine instructions. The default memory range is the function surrounding the program counter of the selected frame. A single argument to this command is a program counter value; GDB dumps the function surrounding this value. Two arguments specify a range of addresses (first inclusive, second exclusive) to dump.

We can use **disassemble** to inspect the object code range shown in the last **info line** example (the example shows SPARC machine instructions):

```
(gdb) disas 0x63e4 0x6404
```

```
Dump of assembler code from 0x63e4 to 0x6404:
0x63e4 <builtin_init+5340>:    ble 0x63f8 <builtin_init+5360>
0x63e8 <builtin_init+5344>:    sethi %hi(0x4c00), %o0
0x63ec <builtin_init+5348>:    ld [%i1+4], %o0
0x63f0 <builtin_init+5352>:    b 0x63fc <builtin_init+5364>
0x63f4 <builtin_init+5356>:    ld [%o0+4], %o0
0x63f8 <builtin_init+5360>:    or %o0, 0x1a4, %o0
0x63fc <builtin_init+5364>:    call 0x9288 <path_search>
0x6400 <builtin_init+5368>:    nop
End of assembler dump.
```

set assembly-language *instruction-set*

This command selects the instruction set to use when disassembling the program via the `disassemble` or `x/i` commands. It is useful for architectures that have more than one native instruction set.

Currently it is only defined for the Intel x86 family. You can set *instruction-set* to either `i386` or `i8086`. The default is `i386`.

8. Examining Data

The usual way to examine data in your program is with the `print` command (abbreviated `p`), or its synonym `inspect`. It evaluates and prints the value of an expression of the language your program is written in.

print *exp*

print /*f* *exp* *exp* is an expression (in the source language). By default the value of *exp* is printed in a format appropriate to its data type; you can choose a different format by specifying /*f*, where *f* is a letter specifying the format.

print

print /*f* If you omit *exp*, GDB displays the last value again (from the *value history*;). This allows you to conveniently inspect the same value in an alternative format.

A more low-level way of examining data is with the `x` command. It examines data in memory at a specified address and prints it in a specified format. *Examining memory*.

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

8.1 Expressions

`print` and many other GDB commands accept an expression and compute its value. Any kind of constant, variable or operator defined by the programming language you are using is valid 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` commands.

GDB now supports array constants in expressions input by the user. The syntax is `{element, element . . . }`. For example, you can now use the command `print {1, 2, 3@}` to build up an array in memory that is malloc'd in the target program.

Because C is so widespread, most of the expressions shown in examples in this manual are in C. See *Using GDB with Different Languages* for information on how to use expressions in other languages.

In this section, we discuss operators that you can use in GDB expressions regardless of your programming language.

Casts are supported in all languages, not just in C, because it is so useful to cast a number into a pointer in order to examine a structure at that address in memory.

GDB supports these operators, in addition to those common to programming languages:

@ **@** is a binary operator for treating parts of memory as arrays. *Artificial arrays*, for more information.

:: **::** allows you to specify a variable in terms of the file or function where it is defined. *Program variables*.

{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 binary operators, just as in a cast). This construct is allowed regardless of what kind of data is normally supposed to reside at *addr*.

8.2 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; they must be either:

- global (or static)
- visible according to the scope rules of the programming language from the point of execution in that frame

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

you can examine and use the variable `a` whenever your program is executing within the function `foo`, but you can only use or examine the variable `b` while your program is executing inside the block where `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 `::` 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.

8.3 Artificial arrays

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

You can do this by referring to a contiguous span of memory as an *artificial array*, using the binary operator `@`. The left operand of `@` should be the first element of the desired array and be an individual object. The right operand should be the desired 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. Here is an 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. Artificial arrays most often appear in expressions via the value history, after printing one out.

Another way to create an artificial array is to use a cast. This re-interprets a value as if it were an array. The value need not be in memory:

```
(gdb) p/x (short[2])0x12345678
$1 = {0x1234, 0x5678}
```

As a convenience, if you leave the array length out (as in **(type)[]value**) gdb calculates the size to fill the value as **sizeof(value)/sizeof(type)**:

```
(gdb) p/x (short[])0x12345678
$2 = {0x1234, 0x5678}
```

Sometimes the artificial array mechanism is not quite enough; in moderately complex data structures, 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 via **RET**. 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
RET
RET
...
```

8.4 Output formats

By default, GDB prints a value according to its data type. Sometimes this is not what you want. For example, you might want to print a number in hex, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or as an instruction. To do these things, specify an *output format* when you print a value.

The simplest use of output formats is to say 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:

- 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. The letter **t** stands for “two”. (**b** cannot be used because these format letters are also used with the **x** command, where **b** stands for “byte”.)
- a** Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located:

```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```
- c** Regard as an integer and print it 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 hex, type

`p/x $pc`

Note that no space is required before the slash; this is because command names in GDB cannot 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 hex.

8.5 Examining memory

You can use the command `x` (for “examine”) to examine memory in any of several formats, independently of your program’s data types.

`x/nfu addr`

`x addr`

`x` Use the `x` command to examine 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 defaults for `nfu`, you need not type the slash `/`. Several commands set convenient defaults for `addr`.

`n`, the repeat count

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

The display format is one of the formats used by `print`, `s` (null-terminated string), or `i` (machine instruction). The default is `x` (hexadecimal) initially. The default changes each time you use either `x` or `print`.

`u`, the unit size

The unit size is any of

- b** Bytes.
- h** Halfwords (two bytes).
- w** Words (four bytes). This is the initial default.
- g** Giant words (eight bytes).

Each time you specify a unit size with `x`, that size becomes the default unit the next time you use `x`. (For the `s` and `i` formats, the unit size is ignored and is normally not written.)

`addr`, starting display address

`addr` is the address where you want GDB to begin displaying memory. The expression need not have a pointer value (though it may); it is always interpreted as an integer address of a byte of memory. *Expressions*, for more information on expressions. The default for `addr` is usually just after the last address examined—but several other commands also set the default address: `info breakpoints` (to the address of the last breakpoint listed), `info line` (to the starting address of a line), and `print` (if you use it to display a value from memory).

For example, `x/3uh 0x54320` is a request to display three halfwords (`h`) of memory, formatted as unsigned decimal integers (`u`), starting at address `0x54320`. `x/4xw $sp` prints the four words (`w`) of memory above the stack pointer (here, `$sp`) in hexadecimal (`x`).

Since the letters indicating unit sizes are all distinct from the letters specifying output formats, you do not have to remember whether unit size or format comes first; either order works. The output specifications `4xw` and `4wx` mean exactly the same thing. (However, the count `n` must come first; `wx4` does not work.)

Even though the unit size *u* is ignored for the formats **s** and **i**, you might still want to use a count *n*; for example, **3i** specifies that you want to see three machine instructions, including any operands. The command **disassemble** gives an alternative way of inspecting machine instructions.

All the defaults for the arguments to **x** are designed to make it easy to continue scanning memory with minimal specifications each time you use **x**. For example, after you have inspected three machine instructions with **x/3i addr**, you can inspect the next seven with just **x/7**. If you use RET to repeat the **x** command, the repeat count *n* is used again; the other arguments default as for successive uses of **x**.

The addresses and contents printed by the **x** command are not saved in the value history because there is 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 **\$__**. 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 is not the same as the last address printed if several units were printed on the last line of output.

8.6 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 prints its value each time your 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
```

This display shows item numbers, expressions and their current values. As with displays you request manually using **x** or **print**, you can specify the output format you prefer; in fact, **display** decides whether to use **print** or **x** depending on how elaborate your format specification is—it uses **x** if you specify a unit size, or one of the two formats (**i** and **s**) that are only supported by **x**; otherwise it uses **print**.

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

display does not repeat if you press RET again after using it.

display/fmt exp

For *fmt* specifying only a display format and not a size or count, add the expression *exp* to the auto-display list but arrange to display it each time in the specified format *fmt*. *Output formats.*

display/fmt addr

For *fmt* **i** or **s**, or including a unit-size or a number of units, add the expression *addr* as a memory address to be examined each time your program stops. Examining means in effect doing **x/fmt addr**.

For example, **display/i \$pc** can be helpful, to see the machine instruction about to be executed each time execution stops (**\$pc** is a common name for the program counter;).

undisplay dnums...

delete display dnums...

Remove item numbers *dnums* from the list of expressions to display.

undisplay does not repeat if you press RET after using it. (Otherwise you

would just get the error **No display number**)

disable display *dnums...*

Disable the display of item numbers *dnums*. A disabled display item is not printed automatically, but is not forgotten. It may be enabled again later.

enable display *dnums...*

Enable display of item numbers *dnums*. It becomes effective once again in auto display of its expression, until you specify otherwise.

display

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

info display

Print the list of expressions previously set up to display automatically, each one with its item number, but without showing the values. This includes disabled expressions, which are marked as such. It also includes expressions which would not be displayed right now because they refer to automatic variables not currently available.

If a display expression refers to local variables, then it does not make sense outside the lexical context for which it was set up. Such an expression is disabled when execution enters a context where one of its variables is not defined. For example, if you give the command **display last_char** while inside a function with an argument **last_char**, GDB displays this argument while your program continues to stop inside that function. When it stops elsewhere—where there is no variable **last_char**—the display is disabled automatically. The next time your program stops where **last_char** is meaningful, you can enable the display expression once again.

8.7 Print settings

GDB provides the following ways to control how arrays, structures, and symbols are printed.

These settings are useful for debugging programs in any language:

set print address

set print address on

GDB prints memory addresses showing the location of stack traces, structure values, pointer values, breakpoints, and so forth, even when it also displays the contents of those addresses. The default is **on**. For example, this is what a stack frame display looks like with **set print address on**:

```
(gdb) f
#0  set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")
    at input.c:530
530      if (lquote != def_lquote)
```

set print address off

Do not print addresses when displaying their contents. For example, this is the same stack frame displayed with **set print address off**:

```
(gdb) set print addr off
(gdb) f
#0  set_quotes (lq="<<", rq=">>") at input.c:530
530      if (lquote != def_lquote)
```

You can use **set print address off** to eliminate all machine dependent displays from the GDB interface. For example, with **print address off**, you should get the same text for backtraces on all machines—whether or not they involve pointer arguments.

show print address

Show whether or not addresses are to be printed.

When GDB prints a symbolic address, it normally prints the closest earlier symbol plus an offset. If that symbol does not uniquely identify the address (for example, it is a name whose scope is a single source file), you may need to clarify. One way to do this is with `info line`, for example `info line *0x4537`. Alternately, you can set GDB to print the source file and line number when it prints a symbolic address:

set print symbol-filename on

Tell GDB to print the source file name and line number of a symbol in the symbolic form of an address.

set print symbol-filename off

Do not print source file name and line number of a symbol. This is the default.

show print symbol-filename

Show whether or not GDB will print the source file name and line number of a symbol in the symbolic form of an address.

Another situation where it is helpful to show symbol filenames and line numbers is when disassembling code; GDB shows you the line number and source file that corresponds to each instruction.

Also, you may wish to see the symbolic form only if the address being printed is reasonably close to the closest earlier symbol:

set print max-symbolic-offset *max-offset*

Tell GDB to only display the symbolic form of an address if the offset between the closest earlier symbol and the address is less than *max-offset*. The default is 0, which tells GDB to always print the symbolic form of an address if any symbol precedes it.

show print max-symbolic-offset

Ask how large the maximum offset is that GDB prints in a symbolic address.

If you have a pointer and you are not sure where it points, try **set print symbol-filename on**. Then you can determine the name and source file location of the variable where it points, using **p/a pointer**. This interprets the address in symbolic form. For example, here GDB shows that a variable `ptt` points at another variable `t`, defined in `hi2.c`:

```
(gdb) set print symbol-filename on
(gdb) p/a ptt
$4 = 0xe008 <t in hi2.c>
```

Warning: For pointers that point to a local variable, **p/a** does not show the symbol name and filename of the referent, even with the appropriate `set print` options turned on.

Other settings control how different kinds of objects are printed:

set print array**set print array on**

Pretty print arrays. This format is more convenient to read, but uses more space. The default is off.

set print array off

Return to compressed format for arrays.

show print array

Show whether compressed or pretty format is selected for displaying arrays.

set print elements *number-of-elements*

Set a limit on how many elements of an array GDB will print. If GDB is printing a large array, it stops printing after it has printed the number of elements set by the `set print elements` command. This limit also applies to the display of strings. Setting *number-of-elements* to zero means that the printing is unlimited.

show print elements

Display the number of elements of a large array that GDB will print. If the number is 0, then the printing is unlimited.

set print null-stop

Cause GDB to stop printing the characters of an array when the first NULL is encountered. This is useful when large arrays actually contain only short strings.

set print pretty on

Cause GDB to print structures in an indented format with one member per line, like this:

```
$1 = {
  next = 0x0,
  flags = {
    sweet = 1,
    sour = 1
  },
  meat = 0x54 "Pork"
}
```

set print pretty off

Cause GDB to print structures in a compact format, like this:

```
$1 = {next = 0x0, flags = {sweet = 1, sour = 1}, meat = 0x54 "Pork"}
```

This is the default format.

show print pretty

Show which format GDB is using to print structures.

set print sevenbit-strings on

Print using only seven-bit characters; if this option is set, GDB displays any eight-bit characters (in strings or character values) using the notation `\nnn`. This setting is best if you are working in English (ascii) and you use the high-order bit of characters as a marker or “meta” bit.

set print sevenbit-strings off

Print full eight-bit characters. This allows the use of more international character sets, and is the default.

show print sevenbit-strings

Show whether or not GDB is printing only seven-bit characters.

set print union on

Tell GDB to print unions which are contained in structures. This is the default setting.

set print union off

Tell GDB not to print unions which are contained in structures.

show print union

Ask GDB whether or not it will print unions which are contained in structures.

For example, given the declarations

```
typedef enum {Tree, Bug} Species;
```

```
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly}
               Bug_forms;

struct thing {
    Species it;
    union {
        Tree_forms tree;
        Bug_forms bug;
    } form;
};

struct thing foo = {Tree, {Acorn}};
```

with `set print union on` in effect `p foo` would print

```
$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}
```

and with `set print union off` in effect it would print

```
$1 = {it = Tree, form = {...}}
```

These settings are of interest when debugging C++ programs:

set print demangle

set print demangle on

Print C++ names in their source form rather than in the encoded (“mangled”) form passed to the assembler and linker for type-safe linkage. The default is **on**.

show print demangle

Show whether C++ names are printed in mangled or demangled form.

set print asm-demangle

set print asm-demangle on

Print C++ names in their source form rather than their mangled form, even in assembler code printouts such as instruction disassemblies. The default is off.

show print asm-demangle

Show whether C++ names in assembly listings are printed in mangled or demangled form.

set demangle-style *style*

Choose among several encoding schemes used by different compilers to represent C++ names. The choices for *style* are currently:

- auto** Allow GDB to choose a decoding style by inspecting your program.
- gnu** Decode based on the GNU C++ compiler (`g++`) encoding algorithm. This is the default.
- lucid** Decode based on the Lucid C++ compiler (`lcc`) encoding algorithm.
- arm** Decode using the algorithm in the *C++ Annotated Reference Manual*. **Warning:** this setting alone is not sufficient to allow debugging `cfront`-generated executables. GDB would require further enhancement to permit that.
- foo** Show the list of formats.

show demangle-style

Display the encoding style currently in use for decoding C++ symbols.

set print object

set print object on

When displaying a pointer to an object, identify the **actual** (derived) type of the object rather than the **declared** type, using the virtual function table.

set print object off

Display only the declared type of objects, without reference to the virtual function table. This is the default setting.

show print object

Show whether actual, or declared, object types are displayed.

set print static-members**set print static-members on**

Print static members when displaying a C++ object. The default is on.

set print static-members off

Do not print static members when displaying a C++ object.

show print static-members

Show whether C++ static members are printed, or not.

set print vtbl**set print vtbl on**

Pretty print C++ virtual function tables. The default is off.

set print vtbl off

Do not pretty print C++ virtual function tables.

show print vtbl

Show whether C++ virtual function tables are pretty printed, or not.

8.8 Value history

Values printed by the `print` command are saved in the GDB *value history*. This allows you to refer to them in other expressions. Values are kept until the symbol table is re-read or discarded (for example with the `file` or `symbol-file` commands). When the symbol table changes, the value history is discarded, since the values may contain pointers back to the types defined in the symbol table.

The values printed are given *history numbers* by which you can refer to them. These are successive integers starting with one. `print` shows you the history number assigned to a value by printing `$num =` before the value; here *num* is the history number.

To refer to any previous value, use `$` followed by the value's history number. The way `print` labels its output is designed to remind you of this. Just `$` refers to the most recent value in the history, and `$$` refers to the value before that. `$$n` refers to the *n*th value from the end; `$$2` is the value just prior to `$$`, `$$1` is equivalent to `$$`, and `$$0` is equivalent to `$`.

For example, suppose you have just printed a pointer to a structure and want to see the contents of the structure. It suffices 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 this:

```
p *$.next
```

You can print successive links in the chain by repeating this command—which you can do by just typing RET.

Note that the history records values, not expressions. If the value of `x` is 4 and you type these commands:

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

then the value recorded in the value history by the `print` command remains 4 even though the value of `x` has changed.

show values

Print the last ten values in the value history, with their item numbers. This is like `p@ $$9` repeated ten times, except that `show values` does not change the history.

show values *n*

Print ten history values centered on history item number *n*.

show values +

Print ten history values just after the values last printed. If no more values are available, `show values +` produces no display.

Pressing RET to repeat `show values n` has exactly the same effect as `show values +`.

8.9 Convenience variables

GDB provides *convenience variables* that you can use within GDB to hold on to a value and refer to it later. These variables exist entirely within GDB; they are not part of your program, and setting a convenience variable has no direct effect on further execution of your program. That is why you can use them freely.

Convenience variables are prefixed with `$`. Any name preceded by `$` can be used for a convenience variable, unless it is one of the predefined machine-specific register names. (Value history references, in contrast, are **numbers** preceded by `$`. *Value history*.)

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`.

Using a convenience variable for the first time creates it, but its value is `void` until you assign a new 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, including structures and arrays, even if that variable already has a value of a different type. The convenience variable, when used as an expression, has the type of its current value.

show convenience

Print a list of convenience variables used so far, and their values. Abbreviated `show con`.

One of the ways to use a convenience variable is as a counter to be incremented or a pointer to be advanced. For example, to print a field from successive elements of an array of structures:

```
set $i = 0
print bar[$i++]->contents
```

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

\$_

The variable `$_` is automatically set by the `x` command to the last address examined. Other commands which provide a default address for `x` to examine also set `$_` to that address; these commands include `info line` and `info breakpoint`. The type of `$_` is `void *` except when set by the `x` command, in which case it is a pointer to the type of `$_`.

- \$__** The variable `$__` is automatically set by the `x` command to the value found in the last address examined. Its type is chosen to match the format in which the data was printed.
- \$_exitcode** The variable `$_exitcode` is automatically set to the exit code when the program being debugged terminates.

8.10 Registers

You can refer to machine register contents, in expressions, as variables with names starting with `$`. The names of registers are different for each machine; use `info registers` to see the names used on your machine.

info registers

Print the names and values of all registers except floating-point registers (in the selected stack frame).

info all-registers

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

info registers *regname ...*

Print the *relativized* value of each specified register *regname*. As discussed in detail below, register values are normally relative to the selected stack frame. *regname* may be any register name valid on the machine you are using, with or without the initial `$`.

GDB has four “standard” register names that are available (in expressions) on most machines—whenever they do not conflict with an architecture’s canonical mnemonics for registers. The register names `$pc` and `$sp` are used for the program counter register and the stack pointer. `$fp` is used for a register that contains a pointer to the current stack frame, and `$ps` is used for a register that contains the processor status. For example, you could print the program counter in hex with

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add four to the stack pointer (This is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected; setting `$sp` is not allowed when other stack frames are selected. To pop entire frames off the stack, regardless of machine architecture, use `return` with

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

Whenever possible, these four standard register names are available on your machine even though the machine has different canonical mnemonics, so long as there is no conflict. The `info registers` command shows the canonical names. For example, on the SPARC, `info registers` displays the processor status register as `$psr` but you can also refer to it as `$ps`.

GDB always considers the contents of an ordinary register as an integer when the register is examined in this way. Some machines have special registers which can hold nothing but floating point; these registers are considered to have floating point values. There is no way to refer to the contents of an ordinary register as floating point value (although you can **print** it as a floating point value with **print/f** *\$regname*).

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 is not the same one that your program normally sees. For example, the registers of the 68881 floating point coprocessor are always saved in “extended” (raw) format, but all C programs expect to work with “double”

(virtual) 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.

Normally, register values are relative to the selected stack 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 true contents of hardware registers, you must select the innermost frame (with **frame 0**).

However, GDB must deduce where registers are saved, from the machine code generated by your compiler. If some registers are not saved, or if GDB is unable to locate the saved registers, the selected stack frame makes no difference.

9. C Language Support

Information specific to the C language is built into GDB so that you can use C expressions while debugging. This also permits GDB to output values in a manner consistent with C conventions.

9.1 C operators

Operators must be defined on values of specific types. For instance, `+` is defined on numbers, but not on structures. Operators are often defined on groups of types. For the purposes of C and C++, the following definitions hold:

- **Integral types** include `int` with any of its storage-class specifiers; `char`; and `enum`.
- **Floating-point types** include `float` and `double`.
- **Pointer types** include all types defined as `(type *)`.
- **Scalar types** include all of the above.

The following operators are supported. They are listed here in order of increasing precedence:

<code>,</code>	The comma or sequencing operator. Expressions in a comma-separated list are evaluated from left to right, with the result of the entire expression being the last expression evaluated.
<code>=</code>	Assignment. The value of an assignment expression is the value assigned. Defined on scalar types.
<code>op=</code>	Used in an expression of the form <code>a op= b</code> , and translated to <code>a = a op b</code> . <i>op= and = have the same precedence. op is any one of the operators <code> </code>, <code>^</code>, <code>&</code>, <code><<</code>, <code>>></code>, <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>.</i>
<code>?:</code>	The ternary operator. <code>a ? b : c</code> can be thought of as: if <i>a</i> then <i>b</i> else <i>c</i> . <i>a</i> should be of an integral type.
<code> </code>	Logical OR. Defined on integral types.
<code>&&</code>	Logical AND. Defined on integral types.
<code> </code>	Bitwise OR. Defined on integral types.
<code>^</code>	Bitwise exclusive-OR. Defined on integral types.
<code>&</code>	Bitwise AND. Defined on integral types.
<code>=, !=</code>	Equality and inequality. Defined on scalar types. The value of these expressions is 0 for false and non-zero for true.
<code><, >, <=, >=</code>	Less than, greater than, less than or equal, greater than or equal. Defined on scalar types. The value of these expressions is 0 for false and non-zero for true.
<code><<, >></code>	left shift, and right shift. Defined on integral types.
<code>@</code>	The GDB “artificial array” operator.
<code>+, -</code>	Addition and subtraction. Defined on integral types, floating-point types and pointer types.
<code>*, /, %</code>	Multiplication, division, and modulus. Multiplication and division are defined on integral and floating-point types. Modulus is defined on integral types.
<code>++, --</code>	Increment and decrement. When appearing before a variable, the operation is performed before the variable is used in an expression; when appearing after it, the variable’s value is used before the operation takes place.

- *** Pointer dereferencing. Defined on pointer types. Same precedence as ++.
- &** Address operator. Defined on variables. Same precedence as ++.
For debugging C++, GDB implements a use of **&** beyond what is allowed in the C++ language itself: you can use **&(&ref)** (or, if you prefer, simply **&&ref**) to examine the address where a C++ reference variable (declared with **&ref**) is stored.
- Negative. Defined on integral and floating-point types. Same precedence as ++.
- !** Logical negation. Defined on integral types. Same precedence as ++.
- ~** Bitwise complement operator. Defined on integral types. Same precedence as ++.
- ., ->** Structure member, and pointer-to-structure member. For convenience, GDB regards the two as equivalent, choosing whether to dereference a pointer based on the stored type information. Defined on **struct** and **union** data.
- []** Array indexing. $a[i]$ is defined as $*(a+i)$. Same precedence as ->.
- ()** Function parameter list. Same precedence as ->.
- ::** C++ scope resolution operator. Defined on **struct**, **union**, and **class** types.
- ::** Doubled colons also represent the GDB scope operator. Same precedence as ::, above.

9.2 C and C++ constants

GDB allows you to express the constants of C and C++ in the following ways:

9.2.1 C constants

GDB allows you to express the constants of C in the following ways:

- Integer constants are a sequence of digits. Octal constants are specified by a leading **0** (i.e. zero), and hexadecimal constants by a leading **0x** or **0X**. Constants may also end with a letter **L**, specifying that the constant should be treated as a **long** value.
- Floating point constants are a sequence of digits, followed by a decimal point, followed by a sequence of digits, and optionally followed by an exponent. An exponent is of the form: **e[+|-]nnn**, where *nnn* is another sequence of digits. The **+** is optional for positive exponents.
- Enumerated constants consist of enumerated identifiers, or their integral equivalents.
- Character constants are a single character surrounded by single quotes (**'**), or a number—the ordinal value of the corresponding character (usually its ASCII value). Within quotes, the single character may be represented by a letter or by *escape sequences*, which are of the form **\nnn**, where *nnn* is the octal representation of the character's ordinal value; or of the form **x**, where *x* is a predefined special character—for example, **\n** for newline.
- String constants are a sequence of character constants surrounded by double quotes (**"**).
- Pointer constants are an integral value. You can also write pointers to constants using the C operator **&**.
- Array constants are comma-separated lists surrounded by braces **{** and **@**; for example, **{1,2,3@}** is a three-element array of integers, **{{1,2@}, {3,4}, {5,6}}** is a three-by-two array, and **{&"hi", &"there", &"fred"@}** is a three-element array of pointers.

9.3 C++ expressions

GDB expression handling has a number of extensions to interpret a significant subset of C++ expressions.

Warning: GDB can only debug C++ code if you compile with the GNU C++ compiler. Moreover, C++ debugging depends on the use of additional debugging information in the symbol table, and thus requires special support. GDB has this support **only** with the stabs debug format. In particular, if your compiler generates a.out, MIPS ECOFF, RS/6000 XCOFF, or ELF with stabs extensions to the symbol table, these facilities are all available. (With GNU CC, you can use the **-gstabs** option to request stabs debugging extensions explicitly.) Where the object code format is standard COFF or DWARF in ELF, on the other hand, most of the C++ support in GDB does **not** work.

1. Member function calls are allowed; you can use expressions like

```
count = aml->GetOriginal(x, y)
```
2. While a member function is active (in the selected stack frame), your expressions have the same namespace available as the member function; that is, GDB allows implicit references to the class instance pointer `this` following the same rules as C++.
3. You can call overloaded functions; GDB resolves the function call to the right definition, with one restriction—you must use arguments of the type required by the function that you want to call. GDB does not perform conversions requiring constructors or user-defined type operators.
4. GDB understands variables declared as C++ references; you can use them in expressions just as you do in C++ source—they are automatically dereferenced.

In the parameter list shown when GDB displays a frame, the values of reference variables are not displayed (unlike other variables); this avoids clutter, since references are often used for large structures. The **address** of a reference variable is always shown, unless you have specified **set print address off**.

5. GDB supports the C++ name resolution operator `::`—your expressions can use it just as expressions in your program do. Since one scope may be defined in another, you can use `::` repeatedly if necessary, for example in an expression like `scope1::scope2::name`. GDB also allows resolving name scope by reference to source files, in both C and C++ debugging.

9.4 C and C++ defaults

If you allow GDB to set type and range checking automatically, they both default to `off` whenever the working language changes to C or C++. This happens regardless of whether you or GDB selects the working language.

If you allow GDB to set the language automatically, it recognizes source files whose names end with `.c`, `.C`, or `.cc`, and when GDB enters code compiled from one of these files, it sets the working language to C or C++. *Having GDB infer the source language*, for further details.

9.5 GDB and C

The `set print union` and `show print union` commands apply to the `union` type. When set to **on**, any `union` that is inside a `struct` or `class` is also printed. Otherwise, it appears as `{...}`.

The `@` operator aids in the debugging of dynamic arrays, formed with pointers and a memory allocation function.

9.6 GDB features for C++

Some GDB commands are particularly useful with C++, and some are designed specifically for use with C++. Here is a summary:

breakpoint menus

When you want a breakpoint in a function whose name is overloaded, GDB breakpoint menus help you specify which function definition you want. *Breakpoint menus.*

rbreak regex

Setting breakpoints using regular expressions is helpful for setting breakpoints on overloaded functions that are not members of any special classes. *Setting breakpoints.*

catch exceptions**info catch**

Debug C++ exception handling using these commands.

pptype typename

Print inheritance relationships as well as other information for type *typename*. *Examining the Symbol Table.*

set print demangle**show print demangle****set print asm-demangle****show print asm-demangle**

Control whether C++ symbols display in their source form, both when displaying code as C++ source and when displaying disassemblies. *Print settings.*

set print object**show print object**

Choose whether to print derived (actual) or declared types of objects. *Print settings.*

set print vtbl**show print vtbl**

Control the format for printing virtual function tables. *Print settings.*

Overloaded symbol names

You can specify a particular definition of an overloaded symbol, using the same notation that is used to declare such symbols in C++: type *symbol (types)* rather than just *symbol*. You can also use the GDB command-line word completion facilities to list the available choices, or to finish the type list for you. See *Command completion* for details on how to do this.

10. Examining the Symbol Table

The commands described in this section allow you to inquire about the symbols (names of variables, functions and types) defined in your program. This information is inherent in the text of your program and does not change as your program executes. GDB finds it in your program's symbol table, in the file indicated when you started GDB, or by one of the file-management commands.

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 the three words **foo . 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**.

info address *symbol*

Describe where the data for *symbol* is stored. For a register variable, this says which register it is kept in. For a non-register local variable, this prints the stack-frame offset at which the variable is always stored.

Note the contrast with **print &symbol**, which does not work at all for a register variable, and for a stack local variable prints the exact address of the current instantiation of the variable.

whatis *exp* Print the data type of expression *exp*. *exp* is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place. *Expressions*.

whatis Print the data type of \$, the last value in the value history.

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*.

ptype *exp* **ptype**

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.

For example, for this variable declaration:

```
struct complex {double real; double imag;} v;
```

the two commands give this output:

```
(gdb) whatis v
type = struct complex
(gdb) ptype v
type = struct complex {
    double real;
    double imag;
}
```

As with **whatis**, using **ptype** without an argument refers to the type of \$, the last value in the value history.

info types *regex*

info types Print a brief description of all types whose name matches *regex* (or all types in your program, if you supply no argument). Each complete typename is matched as though it were a complete line; thus, **i type value** gives information on all types

in your program whose name includes the string `value`, but `info type ^value$` gives information only on types whose complete name is `value`.

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

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 Print the names of all source files in your 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 functions Print the names and data types of all defined functions.

info functions *regex* Print the names and data types of all defined functions whose names contain a match for regular expression *regex*. Thus, **info fun step** finds all functions whose names include `step`; **info fun ^step** finds those whose names start with `step`.

info variables Print the names and data types of all variables that are declared outside of functions (i.e., excluding local variables).

info variables *regex* Print the names and data types of all variables (except for local variables) whose names contain a match for regular expression *regex*.

Some systems allow individual object files that make up your program to be replaced without stopping and restarting your program. If you are running on one of these systems, you can allow GDB to reload the symbols for automatically relinked modules:

set symbol-reloading on Replace symbol definitions for the corresponding source file when an object file with a particular name is seen again.

set symbol-reloading off Do not replace symbol definitions when re-encountering object files of the same name. This is the default state; if you are not running on a system that permits automatically relinking modules, you should leave `symbol-reloading` off, since otherwise GDB may discard symbols when linking large programs, that may contain several modules (from different directories or libraries) with the same name.

show symbol-reloading Show the current `on` or `off` setting.

maint print symbols *filename*

maint print psymbols *filename*

maint print msymbols *filename*

Write a dump of debugging symbol data into the file *filename*. These commands are used to debug the GDB symbol-reading code. Only symbols with debugging data are included. If you use `maint print symbols`, GDB includes all the symbols for which it has already collected full details; that is, *filename* reflects symbols for only those files whose symbols GDB has read. You can use the command `info sources` to find out which files these are. If you use **maint**

print psymbols instead, the dump shows information about symbols that GDB only knows partially—that is, symbols defined in files that GDB has skimmed, but not yet read completely. Finally, **maint print msymbols** dumps just the minimal symbol information required for each object file from which GDB has read some symbols. *Commands to specify files*, for a discussion of how GDB reads symbols (in the description of `symbol-file`).

11. Altering Execution

Once you think you have found an error in your program, you might want to find out for certain whether correcting the apparent error would lead to correct results in the rest of the run. You can find the answer by experiment, using the GDB features for altering execution of the program.

For example, you can store new values into variables or memory locations, give your program a signal, restart it at a different address, or even return prematurely from a function.

11.1 Assignment to variables

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

```
print x=4
```

stores the value 4 into the variable `x`, and then prints the value of the assignment expression (which is 4). *Using GDB with Different Languages*, for more information on operators in supported languages.

If you are not interested in seeing the value of the assignment, use the `set` command instead of the `print` command. `set` is really the same as `print` except that the expression's value is not printed and is not put in the value history. The expression is evaluated only for its effects.

If the beginning of the argument string of the `set` command appears identical to a `set` subcommand, use the `set variable` command instead of just `set`. This command is identical to `set` except for its lack of subcommands. For example, if your program has a variable `width`, you get an error if you try to set a new value with just `set width=13`, because GDB has the command `set width`:

```
(gdb) whatis width
type = double
(gdb) p width
$4 = 13
(gdb) set width=47
Invalid syntax in expression.
```

The invalid expression, of course, is `=47`. In order to actually set the program's variable `width`, use

```
(gdb) set var width=47
```

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

To store values into arbitrary places in memory, use the `...` construct to generate a value of specified type at a specified address. For example, `int@0x83040` refers to memory location `0x83040` as an integer (which implies a certain size and representation in memory), and

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

stores the value 4 into that memory location.

11.2 Continuing at a different address

Ordinarily, when you continue your program, you do so at the place where it stopped, with the `continue` command. You can instead continue at an address of your own choosing, with the following commands:

jump *linespec*

Resume execution at line *linespec*. Execution stops again immediately if there is a breakpoint there. See *Printing source lines*, for a description of the different forms of *linespec*.

The `jump` command does not 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 line *linespec* is in a different function from the one currently executing, the results may be bizarre 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 is not in the function currently executing. However, even bizarre results are predictable if you are well acquainted with the machine-language code of your program.

jump **address*

Resume execution at the instruction at address *address*.

You can get much the same effect as the `jump` command by storing a new value into the register `$pc`. The difference is that this does not start your program running; it only changes the address of where it **will** run when you continue. For example,

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

makes the next `continue` command or stepping command execute at address 0x485, rather than at the address where your program stopped. *Continuing and stepping*.

The most common occasion to use the `jump` command is to back up—perhaps with more breakpoints set—over a portion of a program that has already executed, in order to examine its execution in more detail.

11.3 Giving your program a 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 zero, 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.

`signal` does not repeat when you press RET a second time after executing the command.

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.

11.4 Returning from a function

return

return *expression*

You can cancel execution of a function call with the `return` command. If you give an *expression* argument, its value is used as the function's return value.

When you use `return`, GDB discards the selected stack frame (and all frames within it). You can think of this as making the discarded frame return prematurely. If you wish to specify a value to be returned, give that value as the argument to `return`.

This pops the selected stack frame (See *Selecting a frame*), and any other frames inside of it, 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 does not resume execution; it leaves the program stopped in the state that would exist if the function had just returned. In contrast, the `finish` command resumes

execution until the selected stack frame returns naturally.

11.5 Calling program functions

call *expr* Evaluate the expression *expr* without displaying `void` returned values.

You can use this variant of the `print` command if you want to execute a function from your program, but without cluttering the output with `void` returned values. If the result is not void, it is printed and saved in the value history.

A new user-controlled variable, *call_scratch_address*, specifies the location of a scratch area to be used when GDB calls a function in the target. This is necessary because the usual method of putting the scratch area on the stack does not work in systems that have separate instruction and data spaces.

11.6 Patching programs

By default, GDB opens the file containing your program's executable code (or the corefile) read-only. This prevents accidental alterations to machine code; but it also prevents you from intentionally patching your program's binary.

If you'd like to be able to patch the binary, you can specify that explicitly with the `set write` command. For example, you might want to turn on internal debugging flags, or even to make emergency repairs.

set write on

set write off If you specify **set write on**, GDB opens executable and core files for both reading and writing; if you specify `set write off` (the default), GDB opens them read-only.

If you have already loaded a file, you must load it again (using the `exec-file` or `core-file` command) after changing `set write`, for your new setting to take effect.

show write Display whether executable files and core files are opened for writing as well as reading.

12. GDB Files

GDB needs to know the file name of the program to be debugged, both in order to read its symbol table and in order to start your program. To debug a core dump of a previous run, you must also tell GDB the name of the core dump file.

12.1 Commands to specify files

You may want to specify executable and core dump file names. The usual way to do this is at start-up time, using the arguments to GDB's start-up commands.

Occasionally it is necessary to change to a different file during a GDB session. Or you may run GDB and forget to specify a file you want to use. In these situations the GDB commands to specify new files are useful.

file *filename*

Use *filename* as the program to be debugged. It is read for its symbols and for the contents of pure memory. It is also the program executed when you use the **run** command. If you do not specify a directory and the file is not found in the GDB working directory, GDB uses the environment variable **PATH** as a list of directories to search, just as the shell does when looking for a program to run. You can change the value of this variable, for both GDB and your program, using the **path** command.

On systems with memory-mapped files, an auxiliary file *filename.syms* may hold symbol table information for *filename*. If so, GDB maps in the symbol table from *filename.syms*, starting up more quickly. See the descriptions of the file options **-mapped** and **-readnow** (available on the command line, and with the commands **file**, **symbol-file**, or **add-symbol-file**, described below), for more information.

file *file* with no argument makes GDB discard any information it has on both executable file and the symbol table.

exec-file [*filename*]

Specify that the program to be run (but not the symbol table) is found in *filename*. GDB searches the environment variable **PATH** if necessary to locate your program. Omitting *filename* means to discard information on the executable file.

symbol-file [*filename*]

Read symbol table information from file *filename*. **PATH** is searched when necessary. Use the **file** command to get both symbol table and program to run from the same file.

symbol-file with no argument clears out GDB information on your program's symbol table.

The **symbol-file** command causes GDB to forget the contents of its convenience variables, the value history, and all breakpoints and auto-display expressions. This is because they may contain pointers to the internal data recording symbols and data types, which are part of the old symbol table data being discarded inside GDB.

symbol-file does not repeat if you press RET again after executing it once.

When GDB is configured for a particular environment, it understands debugging information in whatever format is the standard generated for that environment; you may use either a GNU compiler, or other compilers that adhere to the local conventions. Best results are usually obtained from GNU compilers; for example, using GCC you can generate debugging information for optimized code.

On some kinds of object files, the `symbol-file` command does not normally read the symbol table in full right away. Instead, it scans the symbol table quickly to find which source files and which symbols are present. The details are read later, one source file at a time, as they are needed.

The purpose of this two-stage reading strategy is to make GDB start up faster. For the most part, it is invisible except for occasional pauses while the symbol table details for a particular source file are being read. (The `set verbose` command can turn these pauses into messages if desired.

We have not implemented the two-stage strategy for COFF yet. When the symbol table is stored in COFF format, `symbol-file` reads the symbol table data in full right away.

symbol-file *filename* [**-readnow**] [**-mapped**]

file *filename* [**-readnow**] [**-mapped**]

You can override the GDB two-stage strategy for reading symbol tables by using the **-readnow** option with any of the commands that load symbol table information, if you want to be sure GDB has the entire symbol table available.

If memory-mapped files are available on your system through the `mmap` system call, you can use another option, **-mapped**, to cause GDB to write the symbols for your program into a reusable file. Future GDB debugging sessions map in symbol information from this auxiliary symbol file (if the program has not changed), rather than spending time reading the symbol table from the executable program. Using the **-mapped** option has the same effect as starting GDB with the **-mapped** command-line option.

You can use both options together, to make sure the auxiliary symbol file has all the symbol information for your program.

The auxiliary symbol file for a program called *myprog* is called *myprog.syms*. *Once this file exists (so long as it is newer than the corresponding executable), GDB always attempts to use it when you debug myprog; no special options or commands are needed.*

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

core-file [*filename*]

Specify the whereabouts of a core dump file to be used as the “contents of memory”. Traditionally, core files contain only some parts of the address space of the process that generated them; GDB can access the executable file itself for other parts.

`core-file` with no argument specifies that no core file is to be used.

Note that the core file is ignored when your program is actually running under GDB. So, if you have been running your program and you wish to debug a core file instead, you must kill the subprocess in which the program is running. To do this, use the `kill` command.

load *filename*

Depending on what remote debugging facilities are configured into GDB, the `load` command may be available. Where it exists, it is meant to make *filename* (an executable) available for debugging on the remote system—by downloading, or dynamic linking, for example. `load` also records the *filename* symbol table in GDB, like the `add-symbol-file` command.

If your GDB does not have a `load` command, attempting to execute it gets the error message “You can’t do that when your target is ...”

The file is loaded at whatever address is specified in the executable. For some object file formats, you can specify the load address when you link the program; for other formats, like `a.out`, the object file format specifies a fixed address.

add-symbol-file *filename address*

add-symbol-file *filename address* [**-readnow**] [**-mapped**]

The `add-symbol-file` command reads additional symbol table information from the file *filename*. You would use this command when *filename* has been dynamically loaded (by some other means) into the program that is running. *address* should be the memory address at which the file has been loaded; GDB cannot figure this out for itself. You can specify *address* as an expression.

The symbol table of the file *filename* is added to the symbol table originally read with the `symbol-file` command. You can use the `add-symbol-file` command any number of times; the new symbol data thus read keeps adding to the old. To discard all old symbol data instead, use the `symbol-file` command.

`add-symbol-file` does not repeat if you press RET after using it.

You can use the **-mapped** and **-readnow** options just as with the `symbol-file` command, to change how GDB manages the symbol table information for *filename*.

add-shared-symbol-file

The `add-shared-symbol-file` command can be used only under Harris’ CXUX operating system for the Motorola 88k. GDB automatically looks for shared libraries, however if GDB does not find yours, you can run `add-shared-symbol-file`. It takes no arguments.

section

The `section` command changes the base address of section `SECTION` of the exec file to `ADDR`. This can be used if the exec file does not contain section addresses, (such as in the `a.out` format), or when the addresses specified in the file itself are wrong. Each section must be changed separately. The “info files” command lists all the sections and their addresses.

info files

info target `info files` and `info target` are synonymous; both print the current target, including the names of the executable and core dump files currently in use by GDB, and the files from which symbols were loaded. The command `help target` lists all possible targets rather than current ones.

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

GDB supports SunOS, SVr4, Irix 5, and IBM RS/6000 shared libraries. GDB automatically loads symbol definitions from shared libraries when you use the `run` command, or when you examine a core file. (Before you issue the `run` command, GDB does not understand references to a function in a shared library, however—unless you are debugging a core file).

info share

info sharedlibrary

Print the names of the shared libraries which are currently loaded.

sharedlibrary *regex*

share *regex* Load shared object library symbols for files matching a Unix regular expression. As with files loaded automatically, it only loads shared libraries required by your program for a core file or after typing `run`. If *regex* is omitted all shared libraries

required by your program are loaded.

12.2 Errors reading symbol files

While reading a symbol file, GDB occasionally encounters problems, such as symbol types it does not recognize, or known bugs in compiler output. By default, GDB does not notify you of such problems, since they are relatively common and primarily of interest to people debugging compilers. If you are interested in seeing information about ill-constructed symbol tables, you can either ask GDB to print only one message about each such type of problem, no matter how many times the problem occurs; or you can ask GDB to print more messages, to see how many times the problems occur, with the `set complaints` command.

The messages currently printed, and their meanings, include:

inner block not inside outer block in *symbol*

The symbol information shows where symbol scopes begin and end (such as at the start of a function or a block of statements). This error indicates that an inner scope block is not fully contained in its outer scope blocks.

GDB circumvents the problem by treating the inner block as if it had the same scope as the outer block. In the error message, *symbol* may be shown as “(don't know)” if the outer block is not a function.

block at address out of order

The symbol information for symbol scope blocks should occur in order of increasing addresses. This error indicates that it does not do so.

GDB does not circumvent this problem, and has trouble locating symbols in the source file whose symbols it is reading. (You can often determine what source file is affected by specifying `set verbose on`.)

bad block start address patched

The symbol information for a symbol scope block has a start address smaller than the address of the preceding source line. This is known to occur in the SunOS 4.1.1 (and earlier) C compiler.

GDB circumvents the problem by treating the symbol scope block as starting on the previous source line.

bad string table offset in symbol *n*

Symbol number *n* contains a pointer into the string table which is larger than the size of the string table.

GDB circumvents the problem by considering the symbol to have the name `foo`, which may cause other problems if many symbols end up with this name.

unknown symbol type `0xnn`

The symbol information contains new data types that GDB does not yet know how to read. `0xnn` is the symbol type of the misunderstood information, in hexadecimal.

GDB circumvents the error by ignoring this symbol information. This usually allows you to debug your program, though certain symbols are not accessible. If you encounter such a problem and feel like debugging it, you can debug `gdb` with itself, breakpoint on `complain`, then go up to the function `read_dbx_symtab` and examine `*bufp` to see the symbol.

stub type has NULL name

GDB could not find the full definition for a struct or class.

const/volatile indicator missing (ok if using g++ v1.x), got...

The symbol information for a C++ member function is missing some information that recent versions of the compiler should have output for it.

info mismatch between compiler and debugger

GDB could not parse a type specification output by the compiler.

13. Specifying a Debugging Target

A *target* is the execution environment occupied by your program. Often, GDB runs in the same host environment as your program; in that case, the debugging target is specified as a side effect when you use the `file` or `core` commands. When you need more flexibility—for example, running GDB on a physically separate host, or controlling a standalone system over a serial port or a realtime system over a TCP/IP connection—you can use the `target` command to specify one of the target types configured for GDB.

13.1 Active targets

There are three classes of targets: processes, core files, and executable files. GDB can work concurrently on up to three active targets, one in each class. This allows you to (for example) start a process and inspect its activity without abandoning your work on a core file.

For example, if you execute `gdb a.out`, then the executable file `a.out` is the only active target. If you designate a core file as well—presumably from a prior run that crashed and coredumped—then GDB has two active targets and uses them in tandem, looking first in the corefile target, then in the executable file, to satisfy requests for memory addresses. (Typically, these two classes of target are complementary, since core files contain only a program's read-write memory—variables and so on—plus machine status, while executable files contain only the program text and initialized data.)

When you type `run`, your executable file becomes an active process target as well. When a process target is active, all GDB commands requesting memory addresses refer to that target; addresses in an active core file or executable file target are obscured while the process target is active.

Use the `core-file` and `exec-file` commands to select a new core file or executable target. To specify as a target a process that is already running, use the `attach` command.

13.2 Commands for managing targets

target *type parameters*

Connects the GDB host environment to a target machine or process. A target is typically a protocol for talking to debugging facilities. You use the argument *type* to specify the type or protocol of the target machine.

Further *parameters* are interpreted by the target protocol, but typically include things like device names or host names to connect with, process numbers, and baud rates.

The `target` command does not repeat if you press RET again after executing the command.

help target Displays the names of all targets available. To display targets currently selected, use either `info target` or `info files`.

help target *name*

Describe a particular target, including any parameters necessary to select it.

set gnutarget *args*

GDB uses its own library BFD to read your files. GDB knows whether it is reading an *executable*, a *core*, or a *.o* file, however you can specify the file format with the `set gnutarget` command. Unlike most `target` commands, with `gnutarget` the `target` refers to a program, not a machine.

Warning: To specify a file format with `set gnutarget`, you must know the actual BFD name.

show gnutarget

Use the `show gnutarget` command to display what file format `gnutarget` is set to read. If you have not set `gnutarget`, GDB will determine the file format for each file automatically and `show gnutarget` displays The current BDF target is "auto".

Here are some common targets (available, or not, depending on the GDB configuration):

target exec program

An executable file. **target exec program** is the same as **exec-file program**.

target core filename

A core dump file. **target core filename** is the same as **core-file filename**.

target remote dev

Remote serial target in GDB-specific protocol. The argument *dev* specifies what serial device to use for the connection (e.g. */dev/ttya*). *Remote debugging*. **target remote** now supports the `load` command. This is only useful if you have some other way of getting the stub to the target system, and you can put it somewhere in memory where it won't get clobbered by the download.

Different targets are available on different configurations of GDB; your configuration may have more or fewer targets.

13.3 Choosing target byte order

You can now choose which byte order to use with a target system. Use the `set endian big` and `set endian little` commands. Use the `set endian auto` command to instruct GDB to use the byte order associated with the executable. You can see the current setting for byte order with the `show endian` command.

Warning: Currently, only embedded MIPS configurations support dynamic selection of target byte order.

13.4 Remote debugging

If you are trying to debug a program running on a machine that cannot run GDB in the usual way, it is often useful to use remote debugging. For example, you might use remote debugging on an operating system kernel, or on a small system which does not have a general purpose operating system powerful enough to run a full-featured debugger.

Some configurations of GDB have special serial or TCP/IP interfaces to make this work with particular debugging targets. In addition, GDB comes with a generic serial protocol (specific to GDB, but not specific to any particular target system) which you can use if you write the remote stubs—the code that runs on the remote system to communicate with GDB.

Other remote targets may be available in your configuration of GDB; use `help target` to list them.

14. Controlling GDB

You can alter the way GDB interacts with you by using the `set` command. For commands controlling how GDB displays data, ; other settings are described here.

14.1 Prompt

GDB indicates its readiness to read a command by printing a string called the *prompt*. This string is normally **(gdb)**. You can change the prompt string with the `set prompt` command. For instance, when debugging GDB with GDB, it is useful to change the prompt in one of the GDB sessions so that you can always tell which one you are talking to.

Note: `set prompt` no longer adds a space for you after the prompt you set. This allows you to set a prompt which ends in a space or a prompt that does not.

set prompt *newprompt*

Directs GDB to use *newprompt* as its prompt string henceforth.

show prompt

Prints a line of the form: **Gdb's prompt is:** *your-prompt*

14.2 Command editing

GDB reads its input commands via the *readline* interface. This GNU library provides consistent behavior for programs which provide a command line interface to the user. Advantages are GNU Emacs-style or vi-style inline editing of commands, *CSH*-like history substitution, and a storage and recall of command history across debugging sessions.

You may control the behavior of command line editing in GDB with the command `set`.

set editing

set editing on

Enable command line editing (enabled by default).

set editing off

Disable command line editing.

show editing

Show whether command line editing is enabled.

14.3 Command history

GDB can keep track of the commands you type during your debugging sessions, so that you can be certain of precisely what happened. Use these commands to manage the GDB command history facility.

set history filename *fname*

Set the name of the GDB command history file to *fname*. This is the file where GDB reads an initial command history list, and where it writes the command history from this session when it exits. You can access this list through history expansion or through the history command editing characters listed below. This file defaults to the value of the environment variable `GDBHISTFILE`, or to `./gdb_history` if this variable is not set.

set history save

set history save on

Record command history in a file, whose name may be specified with the `set history filename` command. By default, this option is disabled.

set history save off

Stop recording command history in a file.

set history size *size*

Set the number of commands which GDB keeps in its history list. This defaults to the value of the environment variable HISTSIZE, or to 256 if this variable is not set.

History expansion assigns special meaning to the character **!**.

Since **!** is also the logical not operator in C, history expansion is off by default. If you decide to enable history expansion with the **set history expansion on** command, you may sometimes need to follow **!** (when it is used as logical not, in an expression) with a space or a tab to prevent it from being expanded. The readline history facilities do not attempt substitution on the strings **!=** and **!(**, even when history expansion is enabled.

The commands to control history expansion are:

set history expansion on**set history expansion**

Enable history expansion. History expansion is off by default.

set history expansion off

Disable history expansion.

The readline code comes with more complete documentation of editing and history expansion features. Users unfamiliar with GNU Emacs or vi may wish to read it.

show history**show history filename****show history save****show history size****show history expansion**

These commands display the state of the GDB history parameters. **show history** by itself displays all four states.

show commands

Display the last ten commands in the command history.

show commands *n*

Print ten commands centered on command number *n*.

show commands +

Print ten commands just after the commands last printed.

14.4 Screen size

Certain commands to GDB may produce large amounts of information output to the screen. To help you read all of it, GDB pauses and asks you for input at the end of each page of output. Type **RET** when you want to continue the output, or **q** to discard the remaining output. Also, the screen width setting determines when to wrap lines of output. Depending on what is being printed, GDB tries to break the line at a readable place, rather than simply letting it overflow onto the following line.

Normally GDB knows the size of the screen from the termcap data base together with the value of the **TERM** environment variable and the **stty rows** and **stty cols** settings. If this is not correct, you can override it with the **set height** and **set width** commands:

set height *lpp***show height****set width *cpl*****show width**

These **set** commands specify a screen height of *lpp* lines and a screen width of *cpl* characters. The associated **show** commands display the current settings.

If you specify a height of zero lines, GDB does not pause during output no matter how long the output is. This is useful if output is to a file or to an editor buffer.

Likewise, you can specify **set width 0** to prevent GDB from wrapping its output.

14.5 Numbers

You can always enter numbers in octal, decimal, or hexadecimal in GDB by the usual conventions: octal numbers begin with **0**, decimal numbers end with **.**, and hexadecimal numbers begin with **0x**. Numbers that begin with none of these are, by default, entered in base 10; likewise, the default display for numbers—when no particular format is specified—is base 10. You can change the default base for both input and output with the **set radix** command.

set input-radix *base*

Set the default base for numeric input. Supported choices for *base* are decimal 8, 10, or 16. *base* must itself be specified either unambiguously or using the current default radix; for example, any of

```
set radix 012
set radix 10.
set radix 0xa
```

sets the base to decimal. On the other hand, **set radix 10** leaves the radix unchanged no matter what it was.

set output-radix *base*

Set the default base for numeric display. Supported choices for *base* are decimal 8, 10, or 16. *base* must itself be specified either unambiguously or using the current default radix.

show input-radix

Display the current default base for numeric input.

show output-radix

Display the current default base for numeric display.

14.6 Optional warnings and messages

By default, GDB is silent about its inner workings. If you are running on a slow machine, you may want to use the **set verbose** command. This makes GDB tell you when it does a lengthy internal operation, so you will not think it has crashed.

Currently, the messages controlled by **set verbose** are those which announce that the symbol table for a source file is being read; see **symbol-file** in *Commands to specify files*.

set verbose on

Enables GDB output of certain informational messages.

set verbose off

Disables GDB output of certain informational messages.

show verbose

Displays whether **set verbose** is on or off.

By default, if GDB encounters bugs in the symbol table of an object file, it is silent; but if you are debugging a compiler, you may find this information useful.

set complaints *limit*

Permits GDB to output *limit* complaints about each type of unusual symbols before becoming silent about the problem. Set *limit* to zero to suppress all complaints; set it to a large number to prevent complaints from being suppressed.

show complaints

Displays how many symbol complaints GDB is permitted to produce.

By default, GDB is cautious, and asks what sometimes seems to be a lot of stupid questions to confirm certain commands. For example, if you try to run a program which is already running:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n)
```

If you are willing to unflinchingly face the consequences of your own commands, you can disable this “feature”:

set confirm off

Disables confirmation requests.

set confirm on

Enables confirmation requests (the default).

show confirm

Displays state of confirmation requests.

15. Canned Sequences of Commands

Aside from breakpoint commands (*Breakpoint command lists*), GDB provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files.

15.1 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 commands may accept up to 10 arguments separated by whitespace. Arguments are accessed within the user command via `$arg0...$arg9`. A trivial example:

```
define adder
  print $arg0 + $arg1 + $arg2
adder 1 2 3
```

its three arguments. Note the arguments are text substitutions, so they may reference variables, use complex expressions, or even perform inferior functions calls.

define *commandname*

Define a command named *commandname*. If there is already a command by that name, you are 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. The end of these commands is marked by a line containing `end`.

if Takes a single argument, which is an expression to evaluate. It is followed by a series of commands that are executed only if the expression is true (nonzero). There can then optionally be a line `else`, followed by a series of commands that are only executed if the expression was false. The end of the list is marked by a line containing `end`.

while The syntax is similar to `if`: the command takes a single argument, which is an expression to evaluate, and must be followed by the commands to execute, one per line, terminated by an `end`. The commands are executed repeatedly as long as the expression evaluates to true.

document *commandname*

Document the user-defined command *commandname*, so that it can be accessed by `help`. The command *commandname* must already be defined. This command reads lines of documentation just as `define` reads the lines of the command definition, ending with `end`. After the `document` command is finished, `help` on command *commandname* displays the documentation you have written.

You may use the `document` command again to change the documentation of a command. Redefining the command with `define` does not change the documentation.

help user-defined

List all user-defined commands, with the first line of the documentation (if any) for each.

show user

show user *commandname*

Display the GDB commands used to define *commandname* (but not its documentation). If no *commandname* is given, display the definitions for all user-defined commands.

When user-defined commands are executed, the commands of the definition are not printed. An error in any command stops execution of the user-defined command.

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

15.2 User-defined command hooks

You may define **hooks**, which are a special kind of user-defined command. Whenever you run the command **foo**, if the user-defined command **hook-foo** exists, it is executed (with no arguments) before that command.

In addition, a pseudo-command, **stop** exists. Defining (**hook-stop**) makes the associated commands execute every time execution stops in your program: before breakpoint commands are run, displays are printed, or the stack frame is printed.

For example, to ignore **SIGALRM** signals while single-stepping, but treat them normally during normal execution, you could define:

```
define hook-stop
handle SIGALRM nopass
end

define hook-run
handle SIGALRM pass
end

define hook-continue
handle SIGLARM pass
end
```

You can define a hook for any single-word command in GDB, but not for command aliases; you should define a hook for the basic command name, e.g. **backtrace** rather than **bt**. If an error occurs during the execution of your hook, execution of GDB commands stops and GDB issues a prompt (before the command that you actually typed had a chance to run).

If you try to define a hook which does not match any known command, you get a warning from the **define** command.

15.3 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 does not mean to repeat the last command, as it would from the terminal.

When you start GDB, it automatically executes commands from its *init files*. These are files named **.gdbinit**. GDB reads the init file (if any) in your home directory, then processes command line options and operands, and then reads the init file (if any) in the current working directory. This is so the init file in your home directory can set options (such as **set complaints**) which affect the processing of the command line options and operands. The init files are not executed if you use the **-nx** option; see *Choosing modes*.

You can also request the execution of a command file with the **source** command:

source *filename*

Execute the command file *filename*.

The lines in a command file are executed sequentially. They are not printed as they are 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 are doing omit the messages when called from command files.

15.4 Commands for controlled output

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

echo *text* Print *text*. Nonprinting characters can be included in *text* using C escape sequences, such as `\n` to print a newline. ***No newline is 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 displaying a string with spaces at the beginning or the end, since leading and trailing spaces are otherwise trimmed from all arguments. To print **and foo =**, use the command **echo \ and foo = **.

A backslash at the end of *text* can be used, as in C, to continue the command onto subsequent lines. For example,

```
echo This is some text\nwhich is continued\nonto several lines.\n
```

produces the same output as

```
echo This is some text\n
echo which is continued\n
echo onto several lines.\n
```

output *expression*

Print the value of *expression* and nothing but that value: no newlines, no `$nn =`. The value is not entered in the value history either. *Expressions*, for more information on expressions.

output/fmt *expression*

Print the value of *expression* in format *fmt*. You can use the same formats as for **print**.

printf *string, expressions...*

Print the values of the *expressions* under the control of *string*. The *expressions* are separated by commas and may be either numbers or pointers. Their values are printed as specified by *string*, exactly as if your program were to execute the C subroutine

```
printf (string, expressions...);
```

For example, you can print two values in hex like this:

```
printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
```

The only backslash-escape sequences that you can use in the format string are the simple ones that consist of backslash followed by a letter.

16. Renamed Commands

The following commands were renamed in GDB 4, in order to make the command set as a whole more consistent and easier to use and remember:

OLD COMMAND	NEW COMMAND
add-syms	add-symbol-file
delete environment	unset environment
info convenience	show convenience
info copying	show copying
info directories	show directories
info editing	show commands
info history	show values
info targets	help target
info values	show values
info version	show version
info warranty	show warranty
set/show addressprint	set/show print address
set/show array-max	set/show print elements
set/show arrayprint	set/show print array
set/show asm-demangle	set/show print asm-demangle
set/show caution	set/show confirm
set/show demangle	set/show print demangle
set/show history write	set/show history save
set/show prettyprint	set/show print pretty
set/show screen-height	set/show height
set/show screen-width	set/show width
set/show sevenbit-strings	set/show print sevenbit-strings
set/show unionprint	set/show print union
set/show vtblprint	set/show print vtbl
unset	[No longer an alias for delete]