Debugging with GDB

The GNU Source-Level Debugger

Ninth Edition, for GDB version 6.2

Richard Stallman, Roland Pesch, Stan Shebs, et al.

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 and C++. For more information, see Section 12.4 [Supported languages], page 113. For more information, see Section 12.4.1 [C and C++], page 114.

Support for Modula-2 is partial. For information on Modula-2, see Section 12.4.3 [Modula-2], page 120.

Debugging Pascal programs which use sets, subranges, file variables, or nested functions does not currently work. GDB does not support entering expressions, printing values, or similar features using Pascal syntax.

GDB can be used to debug programs written in Fortran, although it may be necessary to refer to some variables with a trailing underscore.

GDB can be used to debug programs written in Objective-C, using either the Apple/NeXT or the GNU Objective-C runtime.

Free software

GDB is *free software*, protected by the GNU General Public License (GPL). The GPL gives you the freedom to copy or adapt a licensed program—but every person getting a copy also gets with it the freedom to modify that copy (which means that they must get access to the source code), and the freedom to distribute further copies. Typical software companies use copyrights to limit your freedoms; the Free Software Foundation uses the GPL to preserve these freedoms.

Fundamentally, the General Public License is a license which says that you have these freedoms and that you cannot take these freedoms away from anyone else.

Free Software Needs Free Documentation

The biggest deficiency in the free software community today is not in the software—it is the lack of good free documentation that we can include with the free software. Many of our most important programs do not come with free reference manuals and free introductory texts. Documentation is an essential part of any software package; when an important free software package does not come with a free manual and a free tutorial, that is a major gap. We have many such gaps today.

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 commands 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` (see Section 19.4 [Screen size], page 183). Since it is easy to press one ⟨RET⟩ too many in this situation, GDB disables command repetition after any command that generates this sort of display.

Any text from a `#` to the end of the line is a comment; it does nothing. This is useful mainly in command files (see Section 20.3 [Command files], page 191).

The *C-o* binding is useful for repeating a complex sequence of commands. This command accepts the current line, like *RET*, and then fetches the next line relative to the current line from the history for editing.

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_environ
make_abs_section             make_function_type
make_blockvector             make_pointer_type
make_cleanup                 make_reference_type
make_command                 make_symbol_completion_list
(gdb) b make_
```

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

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

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

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

For more information about overloaded functions, see Section 12.4.1.3 [C++ expressions], page 116. You can use the command `set overload-resolution off` to disable overload resolution; see Section 12.4.1.7 [GDB features for C++], page 118.

3.3 Getting help

You can always ask GDB itself for information on its commands, using the command `help`.

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

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without

               stopping the program
user-defined -- User-defined commands

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:
```

```
info -- Generic command for showing things
 about the program being debugged
show -- Generic command for showing things
 about the debugger

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.

`apropos` *args*
: The `apropos` *args* command searches through all of the GDB commands, and their documentation, for the regular expression specified in *args*. It prints out all matches found. For example:

```
apropos reload
```

results in:

```
set symbol-reloading -- Set dynamic symbol table reloading
                                    multiple times in one run
show symbol-reloading -- Show dynamic symbol table reloading
                                    multiple times in one run
```

`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
```

results in:

```
if
ignore
info
inspect
```

This is intended for use by GNU Emacs.

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. See [Index], page 363.

`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 need to determine which version of GDB you are running; as GDB evolves, new commands are introduced, and old ones may wither away. Also, many system vendors ship variant versions of GDB, and there are variant versions of GDB in GNU/Linux distributions as well. The version number is the same as the one announced when you start GDB.

`show copying`
Display information about permission for copying GDB.

`show warranty`
Display the GNU "NO WARRANTY" statement, or a warranty, if your version of GDB comes with one.

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. If you are doing native debugging, 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.

Most compilers do not include information about preprocessor macros in the debugging information if you specify the '`-g`' flag alone, because this information is rather large. Version 3.1 of GCC, the GNU C compiler, provides macro information if you specify the options '`-gdwarf-2`' and '`-g3`'; the former option requests debugging information in the Dwarf 2 format, and the latter requests "extra information". In the future, we hope to find more compact ways to represent macro information, so that it can be included with '`-g`' alone.

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 (except on VxWorks) with an argument to GDB (see Chapter 2 [Getting In and Out of GDB], page 11), or by using the `file` or `exec-file` command (see Section 15.1 [Commands to specify files], page 137).

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. See Section 4.3 [Your program's arguments], page 25.

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. See Section 4.4 [Your program's environment], page 26.

The *working directory.*

Your program inherits its working directory from GDB. You can set the GDB working directory with the `cd` command in GDB. See Section 4.5 [Your program's working directory], page 27.

The *standard input and output.*

Your program normally uses the same device for standard input and standard output as GDB is using. You can redirect input and output in the `run` command line, or you can use the `tty` command to set a different device for your program. See Section 4.6 [Your program's input and output], page 27.

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. See Chapter 5 [Stopping and continuing], page 33, 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. See Chapter 8 [Examining Data], page 67.

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.

`start` The name of the main procedure can vary from language to language. With C or C++, the main procedure name is always `main`, but other languages such as Ada do not require a specific name for their main procedure. The debugger provides a convenient way to start the execution of the program and to stop at the beginning of the main procedure, depending on the language used.

The '`start`' command does the equivalent of setting a temporary breakpoint at the beginning of the main procedure and then invoking the '`run`' command.

Some programs contain an elaboration phase where some startup code is executed before the main program is called. This depends on the languages used to write your program. In C++ for instance, constructors for static and global objects are executed before `main` is called. It is therefore possible that the debugger stops before reaching the main procedure. However, the temporary breakpoint will remain to halt execution.

Specify the arguments to give to your program as arguments to the '`start`' command. These arguments will be given verbatim to the underlying '`run`' command. Note that the same arguments will be reused if no argument is provided during subsequent calls to '`start`' or '`run`'.

It is sometimes necessary to debug the program during elaboration. In these cases, using the `start` command would stop the execution of your program too late, as the program would have already completed the elaboration phase. Under these circumstances, insert breakpoints in your elaboration code before running your program.

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 the default shell ('`/bin/sh`' on Unix).

On non-Unix systems, the program is usually invoked directly by GDB, which emulates I/O redirection via the appropriate system calls, and the wildcard characters are expanded by the startup code of the program, not by the shell.

`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) that will be passed to your program. The value of `PATH` used by GDB does not change. You may specify several directory names, separated by whitespace or by a system-dependent separator character (‘`:`’ on Unix, ‘`;`’ on MS-DOS and MS-Windows). 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 the debugged 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: On Unix systems, GDB runs your program using the shell indicated by your `SHELL` environment variable if it exists (or `/bin/sh` if not). If your `SHELL` variable names a

shell that runs an initialization file—such as '`.cshrc`' for C-shell, or '`.bashrc`' for BASH—any variables you set in that file affect your program. You may wish to move setting of environment variables to files that are only run when you sign on, such as '`.login`' or '`.profile`'.

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. See Section 15.1 [Commands to specify files], page 137.

`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 you use `attach`, the debugger finds the program running in the process first by looking in the current working directory, then (if the program is not found) by using the source file search path (see Section 7.4 [Specifying source directories], page 63). You can also use the `file` command to load the program. See Section 15.1 [Commands to Specify Files], page 137.

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

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

If you exit GDB or use the `run` command while you have an attached process, you kill that process. By default, GDB asks for confirmation if you try to do either of these things; you can control whether or not you need to confirm by using the `set confirm` command (see Section 19.7 [Optional warnings and messages], page 185).

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

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 catchpoints

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 (see Section 5.1.1 [Setting breakpoints], page 34), to specify the place where your program should stop by line number, function name or exact address in the program.

In HP-UX, SunOS 4.x, SVR4, and Alpha OSF/1 configurations, you can set breakpoints in shared libraries before the executable is run. There is a minor limitation on HP-UX systems: you must wait until the executable is run in order to set breakpoints in shared library routines that are not called directly by the program (for example, routines that are arguments in a `pthread_create` call).

A *watchpoint* is a special breakpoint that stops your program when the value of an expression changes. You must use a different command to set watchpoints (see Section 5.1.2 [Setting watchpoints], page 37), but aside from that, you can manage a watchpoint like any other breakpoint: you enable, disable, and delete both breakpoints and watchpoints using the same commands.

You can arrange to have values from your program displayed automatically whenever GDB stops at a breakpoint. See Section 8.6 [Automatic display], page 72.

A *catchpoint* is another special breakpoint that stops your program when a certain kind of event occurs, such as the throwing of a C++ exception or the loading of a library. As with watchpoints, you use a different command to set a catchpoint (see Section 5.1.3 [Setting catchpoints], page 39), but aside from that, you can manage a catchpoint like any other breakpoint. (To stop when your program receives a signal, use the `handle` command; see Section 5.3 [Signals], page 50.)

GDB assigns a number to each breakpoint, watchpoint, or catchpoint 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.

Some GDB commands accept a range of breakpoints on which to operate. A breakpoint range is either a single breakpoint number, like '`5`', or two such numbers, in increasing order, separated by a hyphen, like '`5-7`'. When a breakpoint range is given to a command, all breakpoint in that range are operated on.

5.1.1 Setting breakpoints

Breakpoints are set with the `break` command (abbreviated `b`). The debugger convenience variable '`$bpnum`' records the number of the breakpoint you've set most recently; see Section 8.9 [Convenience variables], page 79, for a discussion of what you can do with convenience variables.

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

`break` *function*
: Set a breakpoint at entry to function *function*. When using source languages that permit overloading of symbols, such as C++, *function* may refer to more than one possible place to break. See Section 5.1.8 [Breakpoint menus], page 45, 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 *stack frame*. (See Section 6.1 [Frames], page 55, for a description of stack frames.)

`break` *linenum*
: Set a breakpoint at line *linenum* in the current source file. The current source file is the last file whose source text was printed. The breakpoint will stop your program just before it executes any of the code on that line.

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

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

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

`break`
: When called without any arguments, `break` sets a breakpoint at the next instruction to be executed in the selected stack frame (see Chapter 6 [Examining the Stack], page 55). In any selected frame but the innermost, this makes your program stop as soon as control returns to that frame. This is similar to the effect of a `finish` command in the frame inside the selected frame—except that `finish` does not leave an active breakpoint. If you use `break` without an argument in the innermost frame, GDB stops the next time it reaches the current location; this may be useful inside loops.

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

`break ... if` *cond*
: Set a breakpoint with condition *cond*; evaluate the expression *cond* each time the breakpoint is reached, and stop only if the value is nonzero—that is, if *cond* evaluates as true. '`...`' stands for one of the possible arguments described above (or no argument) specifying where to break. See Section 5.1.6 [Break conditions], page 42, for more information on breakpoint conditions.

`tbreak` *args*
: Set a breakpoint enabled only for one stop. *args* are the same as for the `break` command, and the breakpoint is set in the same way, but the breakpoint is automatically deleted after the first time your program stops there. See Section 5.1.5 [Disabling breakpoints], page 41.

`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 SPARClite DSU and some x86-based targets. These targets 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 take a limited number of breakpoints. For example, on the DSU, only two data breakpoints can be set at a time, and GDB will reject this command if more than two are used. Delete or disable unused hardware breakpoints before setting new ones (see Section 5.1.5 [Disabling], page 41). See Section 5.1.6 [Break conditions], page 42. See [set remote hardware-breakpoint-limit], page 156.

`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. See Section 5.1.5 [Disabling breakpoints], page 41. See also Section 5.1.6 [Break conditions], page 42.

`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.

The syntax of the regular expression is the standard one used with tools like '`grep`'. Note that this is different from the syntax used by shells, so for instance `foo*` matches all functions that include an `fo` followed by zero or more `o`s. There is an implicit `.*` leading and trailing the regular expression you supply, so to match only functions that begin with `foo`, use `^foo`.

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, watchpoints, and catchpoints set and not deleted, with the following columns for each breakpoint:

Breakpoint Numbers

Type Breakpoint, watchpoint, or catchpoint.

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. If the breakpoint is pending (see below for details) on a future load of a shared library, the address will be listed as '`<PENDING>`'.

What Where the breakpoint is in the source for your program, as a file and line number. For a pending breakpoint, the original string passed to the breakpoint command will be listed as it cannot be resolved until the appropriate shared library is loaded in the future.

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. A pending breakpoint is allowed to have a condition specified for it. The condition is not parsed for validity until a shared library is loaded that allows the pending breakpoint to resolve to a valid location.

`info break` with a breakpoint number *n* as argument lists only that breakpoint. The convenience variable `$_` and the default examining-address for the `x` command are set to the address of the last breakpoint listed (see Section 8.5 [Examining memory], page 71).

`info break` 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 (see Section 5.1.6 [Break conditions], page 42).

If a specified breakpoint location cannot be found, it may be due to the fact that the location is in a shared library that is yet to be loaded. In such a case, you may want GDB to create a special breakpoint (known as a *pending breakpoint*) that attempts to resolve itself in the future when an appropriate shared library gets loaded.

Pending breakpoints are useful to set at the start of your GDB session for locations that you know will be dynamically loaded later by the program being debugged. When shared libraries are loaded, a check is made to see if the load resolves any pending breakpoint locations. If a pending breakpoint location gets resolved, a regular breakpoint is created and the original pending breakpoint is removed.

GDB provides some additional commands for controlling pending breakpoint support:

`set breakpoint pending auto`
: This is the default behavior. When GDB cannot find the breakpoint location, it queries you whether a pending breakpoint should be created.

`set breakpoint pending on`
: This indicates that an unrecognized breakpoint location should automatically result in a pending breakpoint being created.

`set breakpoint pending off`
: This indicates that pending breakpoints are not to be created. Any unrecognized breakpoint location results in an error. This setting does not affect any pending breakpoints previously created.

`show breakpoint pending`
: Show the current behavior setting for creating pending breakpoints.

Normal breakpoint operations apply to pending breakpoints as well. You may specify a condition for a pending breakpoint and/or commands to run when the breakpoint is reached. You can also enable or disable the pending breakpoint. When you specify a condition for a pending breakpoint, the parsing of the condition will be deferred until the point where the pending breakpoint location is resolved. Disabling a pending breakpoint tells GDB to not attempt to resolve the breakpoint on any subsequent shared library load. When a pending breakpoint is re-enabled, GDB checks to see if the location is already resolved. This is done because any number of shared library loads could have occurred since the time the breakpoint was disabled and one or more of these loads could resolve the location.

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`' (see [maint info breakpoints], page 303).

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.

Depending on your system, watchpoints may be implemented in software or hardware. GDB does software watchpointing by single-stepping your program and testing the variable's value each time, which is hundreds of times slower than normal execution. (But this may still be worth it, to catch errors where you have no clue what part of your program is the culprit.)

On some systems, such as HP-UX, GNU/Linux and some other x86-based targets, GDB includes support for hardware watchpoints, which do not slow down the running of your program.

`watch` *expr*
: Set a watchpoint for an expression. GDB will break when *expr* is written into by the program and its value changes.

`rwatch` *expr*
: Set a watchpoint that will break when watch *expr* is read by the program.

`awatch` *expr*
: Set a watchpoint that will break when *expr* is either read or written into by the program.

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

GDB sets a *hardware watchpoint* if possible. Hardware watchpoints execute very quickly, and the debugger reports a change in value at the exact instruction where the change occurs. If GDB cannot set a hardware watchpoint, it sets a software watchpoint, which executes more slowly and reports the change in value at the next statement, not the instruction, after the change occurs.

When you issue the `watch` command, GDB reports

```
Hardware watchpoint num: expr
```

if it was able to set a hardware watchpoint.

Currently, the `awatch` and `rwatch` commands can only set hardware watchpoints, because accesses to data that don't change the value of the watched expression cannot be detected without examining every instruction as it is being executed, and GDB does not do that currently. If GDB finds that it is unable to set a hardware breakpoint with the `awatch` or `rwatch` command, it will print a message like this:

```
Expression cannot be implemented with read/access watchpoint.
```

Sometimes, GDB cannot set a hardware watchpoint because the data type of the watched expression is wider than what a hardware watchpoint on the target machine can handle. For example, some systems can only watch regions that are up to 4 bytes wide; on such systems you cannot set hardware watchpoints for an expression that yields a double-precision floating-point number (which is typically 8 bytes wide). As a work-around, it might be possible to break the large region into a series of smaller ones and watch them with separate watchpoints.

If you set too many hardware watchpoints, GDB might be unable to insert all of them when you resume the execution of your program. Since the precise number of active watchpoints is unknown until such time as the program is about to be resumed, GDB might not be

able to warn you about this when you set the watchpoints, and the warning will be printed only when the program is resumed:

`Hardware watchpoint` *num*`: Could not insert watchpoint`

If this happens, delete or disable some of the watchpoints.

The SPARClite 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.

If you call a function interactively using `print` or `call`, any watchpoints you have set will be inactive until GDB reaches another kind of breakpoint or the call completes.

GDB automatically deletes watchpoints that watch local (automatic) variables, or expressions that involve such variables, when they go out of scope, that is, when the execution leaves the block in which these variables were defined. In particular, when the program being debugged terminates, *all* local variables go out of scope, and so only watchpoints that watch global variables remain set. If you rerun the program, you will need to set all such watchpoints again. One way of doing that would be to set a code breakpoint at the entry to the `main` function and when it breaks, set all the watchpoints.

> *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.
>
> *HP-UX Warning:* In multi-thread programs, software watchpoints have only limited usefulness. If GDB creates a software watchpoint, it 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 software watchpoints as usual. However, GDB may not notice when a non-current thread's activity changes the expression. (Hardware watchpoints, in contrast, watch an expression in all threads.)

See [set remote hardware-watchpoint-limit], page 156.

5.1.3 Setting catchpoints

You can use *catchpoints* to cause the debugger to stop for certain kinds of program events, such as C++ exceptions or the loading of a shared library. Use the `catch` command to set a catchpoint.

`catch` *event*

Stop when *event* occurs. *event* can be any of the following:

`throw` The throwing of a C++ exception.

`catch` The catching of a C++ exception.

`exec` A call to `exec`. This is currently only available for HP-UX.

`fork` A call to `fork`. This is currently only available for HP-UX.

`vfork` A call to `vfork`. This is currently only available for HP-UX.

`load`
`load` *libname*
The dynamic loading of any shared library, or the loading of the library *libname*. This is currently only available for HP-UX.

`unload`
`unload` *libname*
The unloading of any dynamically loaded shared library, or the unloading of the library *libname*. This is currently only available for HP-UX.

`tcatch` *event*
Set a catchpoint that is enabled only for one stop. The catchpoint is automatically deleted after the first time the event is caught.

Use the `info break` command to list the current catchpoints.

There are currently some limitations to C++ exception handling (`catch throw` and `catch catch`) 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 either to abort or to simply continue running until it hits a breakpoint, catches a signal that GDB is listening for, or exits. This is the case even if you set a catchpoint for the exception; catchpoints on exceptions are disabled within interactive calls.
- You cannot raise an exception interactively.
- You cannot install an exception handler interactively.

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

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

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

To make the debugger catch all exceptions before any stack unwinding takes place, set a breakpoint on `__raise_exception` (see Section 5.1 [Breakpoints; watchpoints; and exceptions], page 33).

With a conditional breakpoint (see Section 5.1.6 [Break conditions], page 42) 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, watchpoint, or catchpoint 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, watchpoints, or catchpoints 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 (see Section 6.3 [Selecting a frame], page 57). 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`] [*range*`...`]
: Delete the breakpoints, watchpoints, or catchpoints of the breakpoint ranges 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, watchpoint, or catchpoint, 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, watchpoints, and catchpoints 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, watchpoints, and catchpoints if you do not know which numbers to use.

A breakpoint, watchpoint, or catchpoint 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.
- Enabled for deletion. The breakpoint stops your program, but immediately after it does so it is deleted permanently. A breakpoint set with the `tbreak` command starts out in this state.

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

`disable` [`breakpoints`] [*range*`...`]
: 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`] [*range*`...`]
: Enable the specified breakpoints (or all defined breakpoints). They become effective once again in stopping your program.

`enable` [`breakpoints`] `once` *range*`...`
: Enable the specified breakpoints temporarily. GDB disables any of these breakpoints immediately after stopping your program.

`enable` [`breakpoints`] `delete` *range*`...`
: 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` (see Section 5.1.1 [Setting breakpoints], page 34), 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 Section 5.2 [Continuing and stepping], page 47.)

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 (see Section 8.1 [Expressions], page 67). 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 than break conditions for the purpose of performing side effects when a breakpoint is reached (see Section 5.1.7 [Breakpoint command lists], page 44).

Break conditions can be specified when a breakpoint is set, by using '`if`' in the arguments to the `break` command. See Section 5.1.1 [Setting breakpoints], page 34. They can also be changed at any time with the `condition` command.

You can also use the `if` keyword with the `watch` command. The `catch` command does not recognize the `if` keyword; `condition` is the only way to impose a further condition on a catchpoint.

`condition` *bnum expression*
: Specify *expression* as the break condition for breakpoint, watchpoint, or catchpoint 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. If *expression* uses symbols not referenced in the context of the breakpoint, GDB prints an error message:

```
No symbol "foo" in current context.
```

GDB does not actually evaluate *expression* at the time the `condition` command (or a command that sets a breakpoint with a condition, like `break if ...`) is given, however. See Section 8.1 [Expressions], page 67.

`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`. See Section 5.2 [Continuing and stepping], page 47.

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

You could achieve the effect of the ignore count with a condition such as '`$foo-- <= 0`' using a debugger convenience variable that is decremented each time. See Section 8.9 [Convenience variables], page 79.

Ignore counts apply to breakpoints, watchpoints, and catchpoints.

5.1.7 Breakpoint command lists

You can give any breakpoint (or watchpoint or catchpoint) 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, watchpoint, or catchpoint 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. See Section 20.4 [Commands for controlled output], page 192.

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++ and Objective-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)
```

5.1.9 "Cannot insert breakpoints"

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

```
Cannot insert breakpoints.
The same program may be running in another process.
```

When this happens, you have three ways to proceed:

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

A similar message can be printed if you request too many active hardware-assisted breakpoints and watchpoints:

```
Stopped; cannot insert breakpoints.
You may have requested too many hardware breakpoints and watchpoints.
```

This message is printed when you attempt to resume the program, since only then GDB knows exactly how many hardware breakpoints and watchpoints it needs to insert.

When this message is printed, you need to disable or remove some of the hardware-assisted breakpoints and watchpoints, and then continue.

5.1.10 "Breakpoint address adjusted..."

Some processor architectures place constraints on the addresses at which breakpoints may be placed. For architectures thus constrained, GDB will attempt to adjust the breakpoint's address to comply with the constraints dictated by the architecture.

One example of such an architecture is the Fujitsu FR-V. The FR-V is a VLIW architecture in which a number of RISC-like instructions may be bundled together for parallel execution. The FR-V architecture constrains the location of a breakpoint instruction within such a bundle to the instruction with the lowest address. GDB honors this constraint by adjusting a breakpoint's address to the first in the bundle.

It is not uncommon for optimized code to have bundles which contain instructions from different source statements, thus it may happen that a breakpoint's address will be adjusted from one source statement to another. Since this adjustment may significantly alter GDB's breakpoint related behavior from what the user expects, a warning is printed when the breakpoint is first set and also when the breakpoint is hit.

A warning like the one below is printed when setting a breakpoint that's been subject to address adjustment:

```
warning: Breakpoint address adjusted from 0x00010414 to 0x00010410.
```

Such warnings are printed both for user settable and GDB's internal breakpoints. If you see one of these warnings, you should verify that a breakpoint set at the adjusted address will have the desired affect. If not, the breakpoint in question may be removed and other breakpoints may be set which will have the desired behavior. E.g., it may be sufficient to place the breakpoint at a later instruction. A conditional breakpoint may also be useful in some cases to prevent the breakpoint from triggering too often.

GDB will also issue a warning when stopping at one of these adjusted breakpoints:

```
warning: Breakpoint 1 address previously adjusted from 0x00010414
to 0x00010410.
```

When this warning is encountered, it may be too late to take remedial action except in cases where the breakpoint is hit earlier or more frequently than expected.

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 it stops due to a signal, you may want to use `handle`, or use '`signal 0`' to resume execution. See Section 5.3 [Signals], page 50.)

`continue` [*ignore-count*]
`c` [*ignore-count*]
`fg` [*ignore-count*]

Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument *ignore-count* allows you to specify a further number of times to ignore a breakpoint at this location; its effect is like that of `ignore` (see Section 5.1.6 [Break conditions], page 42).

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` (for *foreground*, as the debugged program is deemed to be the foreground program) are provided purely for convenience, and have exactly the same behavior as `continue`.

To resume execution at a different place, you can use `return` (see Section 14.4 [Returning from a function], page 135) to go back to the calling function; or `jump` (see Section 14.2 [Continuing at a different address], page 134) to go to an arbitrary location in your program.

A typical technique for using stepping is to set a breakpoint (see Section 5.1 [Breakpoints; watchpoints; and catchpoints], page 33) 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 only stops at the first instruction of a source line. This prevents the multiple stops that could otherwise occur in `switch` statements, `for` loops, etc. `step` continues to stop if a function that has debugging information is called within the line. In other words, `step` *steps inside* any functions called within the line.

Also, the `step` command only enters a function if there is line number information for the function. 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 only stops at the first instruction of a source line. This prevents multiple stops that could otherwise occur in `switch` statements, `for` loops, etc.

`set step-mode`
`set step-mode on`
: The `set step-mode on` command causes the `step` command to stop at the first instruction of a function which contains no debug line information rather than stepping over it.

  This is useful in cases where you may be interested in inspecting the machine instructions of a function which has no symbolic info and do not want GDB to automatically skip over this function.

`set step-mode off`
: Causes the `step` command to step over any functions which contains no debug information. This is the default.

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

  Contrast this with the `return` command (see Section 14.4 [Returning from a function], page 135).

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

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

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

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

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

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

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

`until` *location*

`u` *location* Continue running your program until either the specified location is reached, or the current stack frame returns. *location* is any of the forms of argument acceptable to `break` (see Section 5.1.1 [Setting breakpoints], page 34). This form of the command uses breakpoints, and hence is quicker than `until` without an argument. The specified location is actually reached only if it is in the current frame. This implies that `until` can be used to skip over recursive function invocations. For instance in the code below, if the current location is line `96`, issuing `until 99` will execute the program up to line `99` in the same invocation of factorial, i.e. after the inner invocations have returned.

```
94 int factorial (int value)
95 {
96     if (value > 1) {
97            value *= factorial (value - 1);
98     }
99     return (value);
100     }
```

`advance` *location*

Continue running the program up to the given location. An argument is required, anything of the same form as arguments for the `break` command. Execution will also stop upon exit from the current stack frame. This command is similar to `until`, but `advance` will not skip over recursive function calls, and the target location doesn't have to be in the same frame as the current one.

`stepi`

`stepi` *arg*

`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. See Section 8.6 [Automatic display], page 72.

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

`nexti`

`nexti` *arg*

`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 character (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* (they kill your program

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 (see Section 6.3 [Selecting a frame], page 57), 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; see Chapter 23 [Using GDB under GNU Emacs], page 203.

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 (see Chapter 6 [Examining the Stack], page 55), this prints lines centered around that line.

`list -`
: Print lines just before the lines last printed.

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

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

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

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

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

`list` *linespec*
: Print lines centered around the line specified by *linespec*.

`list` *first*`,`*last*
: Print lines from *first* to *last*. Both arguments are linespecs.

`list ,`*last* Print lines ending with *last*.

`list` *first*`,`
Print lines starting with *first*.

`list +` Print lines just after the lines last printed.

`list -` Print lines just before the lines last printed.

`list` As described in the preceding table.

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

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

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

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

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

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

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

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

7.2 Editing source files

To edit the lines in a source file, use the `edit` command. The editing program of your choice is invoked with the current line set to the active line in the program. Alternatively, there are several ways to specify what part of the file you want to print if you want to see other parts of the program.

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

`edit` Edit the current source file at the active line number in the program.

`edit` *number*
Edit the current source file with *number* as the active line number.

`edit` *function*
Edit the file containing *function* at the beginning of its definition.

`edit` *filename*`:`*number*
Specifies line *number* in the source file *filename*.

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

`edit *`*address*
: Specifies the line containing the program address *address*. *address* may be any expression.

7.2.1 Choosing your editor

You can customize GDB to use any editor you want[1]. By default, it is /bin/ex, but you can change this by setting the environment variable `EDITOR` before using GDB. For example, to configure GDB to use the `vi` editor, you could use these commands with the `sh` shell:

```
EDITOR=/usr/bin/vi
export EDITOR
gdb ...
```

or in the `csh` shell,

```
setenv EDITOR /usr/bin/vi
gdb ...
```

7.3 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`.

7.4 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

[1] The only restriction is that your editor (say `ex`), recognizes the following command-line syntax:

```
ex +number file
```

The optional numeric value +*number* designates the active line in the file.

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 includes only '`cdir`' and '`cwd`', in that order. 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 '`:`' ('`;`' on MS-DOS and MS-Windows, where '`:`' usually appears as part of absolute file names) 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.

7.5 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 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 (see Section 7.1 [Printing source lines], page 61).

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_changequote
Line 895 of "builtin.c" starts at pc 0x634c and ends at 0x6350.
```

We can also inquire (using **addr* 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 (see Section 8.5 [Examining memory], page 71). Also, this address is saved as the value of the convenience variable `$_` (see Section 8.9 [Convenience variables], page 79).

`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.

The following example shows the disassembly of a range of addresses of HP PA-RISC 2.0 code:

```
(gdb) disas 0x32c4 0x32e4
Dump of assembler code from 0x32c4 to 0x32e4:
0x32c4 <main+204>:      addil 0,dp
0x32c8 <main+208>:      ldw 0x22c(sr0,r1),r26
0x32cc <main+212>:      ldil 0x3000,r31
0x32d0 <main+216>:      ble 0x3f8(sr4,r31)
0x32d4 <main+220>:      ldo 0(r31),rp
0x32d8 <main+224>:      addil -0x800,dp
0x32dc <main+228>:      ldo 0x588(r1),r26
0x32e0 <main+232>:      ldil 0x3000,r31
End of assembler dump.
```

Some architectures have more than one commonly-used set of instruction mnemonics or other syntax.

`set disassembly-flavor` *instruction-set*

Select the instruction set to use when disassembling the program via the `disassemble` or `x/i` commands.

Currently this command is only defined for the Intel x86 family. You can set *instruction-set* to either `intel` or `att`. The default is `att`, the AT&T flavor used by default by Unix assemblers for x86-based targets.

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 (see Chapter 12 [Using GDB with Different Languages], page 109).

`print` *expr*
`print /`*f* *expr*
: *expr* is an expression (in the source language). By default the value of *expr* 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; see Section 8.4 [Output formats], page 70.

`print`
`print /`*f*
: If you omit *expr*, GDB displays the last value again (from the *value history*; see Section 8.8 [Value history], page 78). 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. See Section 8.5 [Examining memory], page 71.

If you are interested in information about types, or about how the fields of a struct or a class are declared, use the `ptype` *exp* command rather than `print`. See Chapter 13 [Examining the Symbol Table], page 127.

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 also includes preprocessor macros, if you compiled your program to include this information; see Section 4.1 [Compilation], page 23.

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

Because C is so widespread, most of the expressions shown in examples in this manual are in C. See Chapter 12 [Using GDB with Different Languages], page 109, 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. See Section 8.3 [Artificial arrays], page 69, for more information.

`::` '`::`' allows you to specify a variable in terms of the file or function where it is defined. See Section 8.2 [Program variables], page 68.

`{`*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 (see Section 6.3 [Selecting a frame], page 57); they must be either:

- global (or file-static)

or

- visible according to the scope rules of the programming language from the point of execution in that frame

This means that in the function

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

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.

This may also happen when the compiler does significant optimizations. To be sure of always seeing accurate values, turn off all optimization when compiling.

Another possible effect of compiler optimizations is to optimize unused variables out of existence, or assign variables to registers (as opposed to memory addresses). Depending on the support for such cases offered by the debug info format used by the compiler, GDB might not be able to display values for such local variables. If that happens, GDB will print a message like this:

```
No symbol "foo" in current context.
```

To solve such problems, either recompile without optimizations, or use a different debug info format, if the compiler supports several such formats. For example, GCC, the GNU C/C++ compiler usually supports the '`-gstabs+`' option. '`-gstabs+`' produces debug info in a format that is superior to formats such as COFF. You may be able to use DWARF 2 ('`-gdwarf-2`'), which is also an effective form for debug info. See section "Options for Debugging Your Program or GNU CC" in *Using GNU CC*.

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 (see Section 8.8 [Value history], page 78), 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 (see Section 8.9 [Convenience variables], page 79) 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".[1]

`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>
```

The command `info symbol 0x54320` yields similar results. See Chapter 13 [Symbols], page 127.

[1] '`b`' cannot be used because these format letters are also used with the `x` command, where '`b`' stands for "byte"; see Section 8.5 [Examining memory], page 71.

`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 (see Section 8.10 [Registers], page 80), 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. See Section 8.1 [Expressions], page 67, 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`'; see Section 8.10 [Registers], page 80) 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; see Section 7.5 [Source and machine code], page 64.

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` *expr*
: Add the expression *expr* to the list of expressions to display each time your program stops. See Section 8.1 [Expressions], page 67.

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

`display/`*fmt expr*
: For *fmt* specifying only a display format and not a size or count, add the expression *expr* to the auto-display list but arrange to display it each time in the specified format *fmt*. See Section 8.4 [Output formats], page 70.

`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*’. See Section 8.5 [Examining memory], page 71.

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; see Section 8.10 [Registers], page 80).

`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.