GDB Manual

The GNU Source-Level Debugger

Third Edition, GDB version 3.2

April 1989

Richard M. Stallman

Copyright © 1988, 1989 Free Software Foundation, Inc.

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

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

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

GNU GENERAL PUBLIC LICENSE

Version 1, February 1989

Copyright © 1989 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The license agreements of most software companies try to keep users at the mercy of those companies. By contrast, our General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. The General Public License applies to the Free Software Foundation's software and to any other program whose authors commit to using it. You can use it for your programs, too.

When we speak of free software, we are referring to freedom, not price. Specifically, the General Public License is designed to make sure that you have the freedom to give away or sell copies of free software, that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of a such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

1. This License Agreement applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any work containing the Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you".
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish

on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this General Public License and to the absence of any warranty; and give any other recipients of the Program a copy of this General Public License along with the Program. You may charge a fee for the physical act of transferring a copy.

3. You may modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:
   - cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and
   - cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License (except that you may choose to grant warranty protection to some or all third parties, at your option).
   - If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the simplest and most usual way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this General Public License.
   - You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

   Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.
4. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:
   - accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

   Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.

5. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Program is void, and will automatically terminate your rights to use the Program under this License. However, parties who have received copies, or rights to use copies, from you under this General Public License will not have their licenses terminated so long as such parties remain in full compliance.
6. By copying, distributing or modifying the Program (or any work based on the Program) you indicate your acceptance of this license to do so, and all its terms and conditions.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.
8. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of the license which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the license, you may choose any version ever published by the Free Software Foundation.
9. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

10. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
11. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT

LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to humanity, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 1, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
program `Gnomovision' (a program to direct compilers to make passes
at assemblers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

That's all there is to it!

1 GDB Input and Output Conventions

GDB is invoked with the shell command '`gdb`'. Once started, it reads commands from the terminal until you tell it to exit.

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

GDB command names may always be abbreviated if the abbreviation is unambiguous. Sometimes 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`'. Possible command abbreviations are often stated in the documentation of the individual commands.

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

A line of input starting with '`#`' is a comment; it does nothing. This is useful mainly in command files (See Section 11.2 [Command Files], page 51).

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

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

To exit GDB, use the '`quit`' command (abbreviated '`q`'). *`Ctrl-c`* will not exit from GDB, but rather will terminate the action of any GDB command that is in progress and return to GDB command level. It is safe to type *`Ctrl-c`* at any time because GDB does not allow it to take effect until a time when it is safe.

Certain commands to GDB may produce large amounts of information output to the screen. To help you read all of it, GDB pauses and asks you for input at the end of each page of output. Type `RET` when you want to continue the output. Normally GDB knows the size of the screen from on the termcap data base together with the value of the `TERM` environment variable; if this is not correct, you can override it with the '`set screensize`' command:

`set screensize` *`lpp`*
`set screensize` *`lpp cpl`*
: Specify a screen height of *lpp* lines and (optionally) a width of *cpl* characters. If you omit *cpl*, the width does not change.

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

Also, GDB may at times produce more information about its own workings than is of interest to the user. Some of these informational messages can be turned on and off with the '`set verbose`' command:

`set verbose off`
: Disables GDB's output of certain informational messages.

`set verbose on`
: Re-enables GDB's output of certain informational messages.

Currently, the messages controlled by '`set verbose`' are those which announce that the symbol table for a source file is being read (see Section 2.2 [File Commands], page 9, in the description of the command '`symbol-file`').

2 Specifying GDB's Files

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

2.1 Specifying Files with Arguments

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

```
gdb progm core
```

specifies `progm` as the executable program and `core` as a core dump file to examine. (You do not need to have a core dump file if what you plan to do is debug the program interactively.)

See Chapter 12 [Options], page 53, for full information on options and arguments for invoking GDB.

2.2 Specifying Files with Commands

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

`exec-file` *filename*
: Specify that the program to be run is found in *filename*. If you do not specify a directory and the file is not found in GDB's working directory, GDB will use the environment variable `PATH` as a list of directories to search, just as the shell does when looking for a program to run.

`symbol-file` *filename*
: Read symbol table information from file *filename*. `PATH` is searched when necessary. Most of the time you will use both the '`exec-file`' and '`symbol-file`' commands on the same file.

  '`symbol-file`' with no argument clears out GDB's symbol table.

  The '`symbol-file`' command does not actually read the symbol table in full right away. Instead, it scans the symbol table quickly to find which source files and which symbols are present. The details are read later, one source file at a time, when they are needed.

  The purpose of this two-stage reading strategy is to make GDB start up faster. For the most part, it is invisible except for occasional messages telling you that the symbol table details for a particular source file are being read. (The '`set verbose`' command controls whether these messages are printed; see Chapter 1 [User Interface], page 7).

  However, you will sometimes see in backtraces lines for functions in source files whose data has not been read in; these lines omit some of the information, such

as argument values, which cannot be printed without full details of the symbol table.

When the symbol table is stored in COFF format, '`symbol-file`' does read the symbol table data in full right away. We haven't bothered to implement the two-stage strategy for COFF.

`core-file` *filename*

Specify the whereabouts of a core dump file to be used as the "contents of memory". Note that the core dump contains only the writable parts of memory; the read-only parts must come from the executable file.

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

Note that the core file is ignored when your program is actually running under GDB. So, if you have been running the program and you wish to debug a core file instead, you must kill the subprocess in which the program is running. To do this, use the '`kill`' command (see Section 4.6 [Kill Process], page 15).

`add-file` *filename address*

The '`add-file`' command reads additional symbol table information from the file *filename*. You would use this when that file has been dynamically loaded into the program that is running. *address* should be the memory address at which the file has been loaded; GDB cannot figure this out for itself.

The symbol table of the file *filename* is added to the symbol table originally read with the '`symbol-file`' command. You can use the '`add-file`' command any number of times; the new symbol data thus read keeps adding to the old. The '`symbol-file`' command forgets all the symbol data GDB has read; that is the only time symbol data is forgotten in GDB.

`info files`

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

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

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

3 Compiling Your Program for Debugging

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

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

The Unix C compiler is unable to handle the '`-g`' and '`-O`' options together. This means that you cannot ask for optimization if you ask for debugger information.

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 the program is correct, but there's no sense in pushing your luck.

GDB no longer supports the debugging information produced by giving the GNU C compiler the '`-gg`' option, so do not use this option.

If your program includes archives made with the `ar` program, and if the object files used as input to `ar` were compiled without the '`-g`' option and have names longer than 15 characters, GDB will get confused reading the program's symbol table. No error message will be given, but GDB may behave strangely. The reason for this problem is a deficiency in the Unix archive file format, which cannot represent file names longer than 15 characters.

To avoid this problem, compile the archive members with the '`-g`' option or use shorter file names. Alternatively, use a version of GNU `ar` dated more recently than August 1989.

4 Running Your Program Under GDB

To start your program under GDB, use the '`run`' command. The program must already have been specified using the '`exec-file`' command or with an argument to GDB (see Chapter 2 [Files], page 9); what '`run`' does is create an inferior process, load the program into it, and set it in motion.

The execution of a program is affected by certain information it receives from its superior. GDB provides ways to specify this information, which you must do *before* starting the program. (You can change it after starting the program, but such changes do not affect the program unless you start it over again.) This information may be divided into three categories:

The *arguments.*
: You specify the arguments to give the program as the arguments of the '`run`' command.

The *environment.*
: The program normally inherits its environment from GDB, but you can use the GDB commands '`set environment`' and '`unset environment`' to change parts of the environment that will be given to the program.

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

After the '`run`' command, the debugger does nothing but wait for your program to stop. See Chapter 5 [Stopping], page 17.

Note that once your program has been started by the '`run`' command, you may evaluate expressions that involve calls to functions in the inferior. See Section 8.1 [Expressions], page 35. If you wish to evaluate a function simply for its side affects, you may use the '`set`' command. See Section 10.1 [Assignment], page 47.

4.1 Your Program's Arguments

The arguments to your program are 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 the program.

'`run`' with no arguments uses the same arguments used by the previous '`run`'.

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

4.2 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 the program with different environments without having to start the debugger over again.

`info environment` *varname*
: Print the value of environment variable *varname* to be given to your program when it is started. This command can be abbreviated '`i env` *varname*'.

`info environment`
: Print the names and values of all environment variables to be given to your program when it is started. This command can be abbreviated '`i env`'.

`set environment` *varname value*
`set environment` *varname* `=` *value*
: Sets environment variable *varname* to *value*, 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. This command can be abbreviated as short as '`set e`'.

  For example, this command:

  ```
  set env USER = foo
  ```

  tells the program, when subsequently run, to assume it is being run on behalf of the user named '`foo`'.

`delete environment` *varname*
`unset environment` *varname*
: Remove variable *varname* from the environment to be passed to your program. This is different from '`set env` *varname* `=`' because '`delete environment`' leaves the variable with no value, which is distinguishable from an empty value. This command can be abbreviated '`d e`'.

4.3 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. GDB's 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 Chapter 2 [Files], page 9.

`cd` *directory*
: Set GDB's working directory to *directory*.

`pwd`
: Print GDB's working directory.

4.4 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.

You can redirect the program's input and/or output using '`sh`'-style redirection commands in the '`run`' command. For example,

```
run > outfile
```

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

Another way to specify where the 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 input/output redirection, 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.

4.5 Debugging an Already-Running Process

Some operating systems allow GDB to debug an already-running process that was started outside of GDB. To do this, you use the '`attach`' command instead of the '`run`' command.

The '`attach`' command requires one argument, which is the process-id of the process you want to debug. (The usual way to find out the process-id of the process is with the `ps` utility.)

The first thing GDB does after arranging to debug the process is to stop it. You can examine and modify an attached process with all the GDB commands that 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.

When you have finished debugging the attached process, you can use the '`detach`' command to release it from GDB's 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`'.

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

The '`attach`' command is also used to debug a remote machine via a serial connection. See Section 4.5 [Attach], page 15, for more info.

4.6 Killing the Child Process

`kill` Kill the child process in which the program being debugged is running under GDB.

This command is useful if you wish to debug a core dump instead. GDB ignores any core dump file if it is actually running the program, so the '`kill`' command is the only sure way to make sure the core dump file is used once again.

It is also useful if you wish to run the program outside the debugger for once and then go back to debugging it.

The '`kill`' command is also useful if you wish to recompile and relink the program, since on many systems it is impossible to modify an executable file

which is running in a process. But, in this case, it is just as good to exit GDB, since you will need to read a new symbol table after the program is recompiled if you wish to debug the new version, and restarting GDB is the easiest way to do that.

5 Stopping and Continuing

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

5.1 Signals

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

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

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

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

`info signal`
: Print a table of all the kinds of signals and how GDB has been told to handle each one. You can use this to see the signal numbers of all the defined types of signals.

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

To use the '`handle`' command you must know the code number of the signal you are concerned with. To find the code number, type '`info signal`' which prints a table of signal names and numbers.

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

`stop`
: GDB should stop the program when this signal happens. This implies the '`print`' keyword as well.

`print`
: GDB should print a message when this signal happens.

`nostop`
: GDB should not stop the program when this signal happens. It may still print a message telling you that the signal has come in.

`noprint`
: GDB should not mention the occurrence of the signal at all. This implies the '`nostop`' keyword as well.

`pass` GDB should allow the program to see this signal; the program will be able to handle the signal, or may be terminated if the signal is fatal and not handled.

`nopass` GDB should not allow the program to see this signal.

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

You can also use the '`signal`' command to prevent the program from seeing a signal, or cause it to see a signal it normally would not see, or to give it any signal at any time. See Section 10.3 [Signaling], page 48.

5.2 Breakpoints

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

Each breakpoint is assigned a number when it is created; these numbers are successive integers starting with 1. In many of the commands for controlling various features of breakpoints you use the breakpoint number to say which breakpoint you want to change. Each breakpoint may be *enabled* or *disabled*; if disabled, it has no effect on the program until you enable it again.

The command '`info break`' prints a list of all breakpoints set and not deleted, showing their numbers, where in the program they are, and any special features in use for them. Disabled breakpoints are included in the list, but marked as disabled. '`info break`' with a breakpoint number 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 8.5.1 [Memory], page 38).

5.2.1 Setting Breakpoints

Breakpoints are set with the '`break`' command (abbreviated '`b`'). You have several ways to say where the breakpoint should go.

`break` *function*
: Set a breakpoint at entry to function *function*.

`break +`*offset*
`break -`*offset*
: Set a breakpoint some number of lines forward or back from the position at which execution stopped in the currently selected frame.

`break` *linenum*
: Set a breakpoint at line *linenum* in the current source file. That file is the last file whose source text was printed. This breakpoint will stop the 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 the program which do not have debugging information or source files.

`break`
: Set a breakpoint at the next instruction to be executed in the selected stack frame (see Chapter 6 [Stack], page 27). In any selected frame but the innermost, this will cause the program to stop as soon as control returns to that frame. This is equivalent to a '`finish`' command in the frame inside the selected frame. If this is done in the innermost frame, GDB will stop the next time it reaches the current location; this may be useful inside of 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 the 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. '`...`' stands for one of the possible arguments described above (or no argument) specifying where to break. See Section 5.2.4 [Conditions], page 21, for more information on breakpoint conditions.

`tbreak` *args*
: Set a breakpoint enabled only for one stop. *args* are the same as in the '`break`' command, and the breakpoint is set in the same way, but the breakpoint is automatically disabled the first time it is hit. See Section 5.2.3 [Disabling], page 20.

GDB allows you to set any number of breakpoints at the same place in the program. There is nothing silly or meaningless about this. When the breakpoints are conditional, this is even useful (see Section 5.2.4 [Conditions], page 21).

5.2.2 Deleting Breakpoints

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

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

It is not necessary to delete a breakpoint to proceed past it. GDB automatically ignores breakpoints in 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 [Selection], page 28). When the innermost frame is selected, this is a good way to delete a breakpoint that the program just stopped at.

`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` *`bnums...`*
Delete the breakpoints of the numbers specified as arguments.

5.2.3 Disabling Breakpoints

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

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

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

You change the state of enablement of a breakpoint with the following commands:

`disable breakpoints` *`bnums...`*
`disable` *`bnums...`*
Disable the specified breakpoints. 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.

`enable breakpoints` *`bnums...`*
`enable` *`bnums...`*
Enable the specified breakpoints. They become effective once again in stopping the program, until you specify otherwise.

`enable breakpoints once` *`bnums...`*
`enable once` *`bnums...`*
Enable the specified breakpoints temporarily. Each will be disabled again the next time it stops the program (unless you have used one of these commands to specify a different state before that time comes).

`enable breakpoints delete` *bnums...*
`enable delete` *bnums...*

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

Aside from the automatic disablement or deletion of a breakpoint when it stops the program, which happens only in certain states, the state of enablement of a breakpoint changes only when one of the commands above is used.

5.2.4 Break Conditions

The simplest sort of breakpoint breaks every time the program reaches a specified place. You can also specify a *condition* for a breakpoint. A condition is just a boolean expression in your programming language. (See Section 8.1 [Expressions], page 35). A breakpoint with a condition evaluates the expression each time the program reaches it, and the program stops only if the condition is true.

Break conditions may have side effects, and may even call functions in your program. These may sound like strange things to do, but their 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 the program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible for the purpose of performing side effects when a breakpoint is reached (see Section 5.2.5 [Break Commands], page 22).

Break conditions can be specified when a breakpoint is set, by using '`if`' in the arguments to the '`break`' command. See Section 5.2.1 [Set Breaks], page 18. They can also be changed at any time with the '`condition`' command:

`condition` *bnum expression*

Specify *expression* as the break condition for breakpoint number *bnum*. From now on, this breakpoint will stop the program only if the value of *expression* is true (nonzero, in C). *expression* is not evaluated at the time the '`condition`' command is given. See Section 8.1 [Expressions], page 35.

`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 the 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 will not stop the next *n* times it is reached.

`ignore` *bnum count*

Set the ignore count of breakpoint number *bnum* to *count*. The next *count* times the breakpoint is reached, it will not stop.

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

`cont` *count*

> Continue execution of the program, setting the ignore count of the breakpoint that the program stopped at to *count* minus one. Thus, the program will not stop at this breakpoint until the *count*'th time it is reached.
>
> This command is allowed only when the program stopped due to a breakpoint. At other times, the argument to '`cont`' is ignored.

If a breakpoint has a positive ignore count and a condition, the condition is not checked. Once the ignore count reaches zero, the condition will start to be checked.

Note that 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 8.8 [Convenience Vars], page 42.

5.2.5 Commands Executed on Breaking

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

`commands` *bnum*

> Specify 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, use the command '`commands`' and follow it immediately by '`end`'; that is, give no commands.
>
> With no arguments, '`commands`' refers to the last breakpoint set.

It is possible for breakpoint commands to start the program up again. Simply use the '`cont`' command, or '`step`', or any other command to resume execution. However, any remaining breakpoint commands are ignored. When the program stops again, GDB will act according to the cause of that stop.

If the first command specified 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 the remaining commands too print nothing, you will see no sign that the breakpoint was reached at all. '`silent`' is not really a command; it is meaningful only at the beginning of the commands for a breakpoint.

The commands '`echo`' and '`output`' that allow you to print precisely controlled output are often useful in silent breakpoints. See Section 11.3 [Output], page 52.

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

```
break foo if x>0
commands
silent
echo x is\040
output x
echo \n
cont
end
```

One application for breakpoint commands is to correct one bug so you can test another. Put a breakpoint just after the erroneous line of code, give it a condition to detect the case in which something erroneous has been done, and give it commands to assign correct values to any variables that need them. End with the '`cont`' command so that the program 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
```

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

In the GNU system, this will be fixed by changing the behavior of terminal modes.

Under Unix, when you have this problem, you might be able to get around it by putting your actions into the breakpoint condition instead of commands. For example

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

specifies a condition expression (See Section 8.1 [Expressions], page 35) that will change `x` as needed, then always have the value 0 so the program will not stop. Loss of input is avoided here because break conditions are evaluated without changing the terminal modes. When you want to have nontrivial conditions for performing the side effects, the operators '`&&`', '`||`' and '`?...:`' may be useful.

5.2.6 "Cannot Insert Breakpoints" Error

Under some operating systems, breakpoints cannot be used in a program if any other process is running that program. Attempting to run or continue the program with a breakpoint in this case will cause GDB to stop it.

When this happens, you have three ways to proceed:

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

5.3 Continuing

After your program stops, most likely you will want it to run some more if the bug you are looking for has not happened yet.

`cont` Continue running the program at the place where it stopped.

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

You can, however, specify an ignore-count for the breakpoint that the program stopped at, by means of an argument to the '`cont`' command. See Section 5.2.4 [Conditions], page 21.

If the program stopped because of a signal other than `SIGINT` or `SIGTRAP`, continuing will cause the program to see that signal. You may not want this to happen. For example, if the program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but the program would probably terminate immediately as a result of the fatal signal once it sees the signal. To prevent this, you can continue with '`signal 0`'. See Section 10.3 [Signaling], page 48. You can also act in advance to prevent the program from seeing certain kinds of signals, using the '`handle`' command (see Section 5.1 [Signals], page 17).

5.4 Stepping

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

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

This command may be given when control is within a function for which there is no debugging information. In that case, execution will proceed until control reaches a different function, or is about to return from this function. An argument repeats this action.

`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` Similar to '`step`', but any function calls appearing within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the stack level which was executing when the '`next`' command was given. This command is abbreviated '`n`'.

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

'`next`' within a function without debugging information acts as does '`step`', but any function calls appearing within the code of the function are executed without stopping.

`finish` Continue running until just after the selected stack frame returns (or until there is some other reason to stop, such as a fatal signal or a breakpoint). Print value returned by the selected stack frame (if any).

Contrast this with the '`return`' command (see Section 10.4 [Returning], page 48).

`until` 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`' will cause the program to continue execution until the loop is exited. In contrast, a '`next`' command at the end of a loop will simply step back to the beginning of the loop, which would force you to step through the next iteration.

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

'`until`' may produce somewhat counterintuitive results if the order of the source lines does not match the actual order of execution. For example, in a typical C `for`-loop, the third expression in the `for`-statement (the loop-step expression) is executed after the statements in the body of the loop, but is written before them. Therefore, the '`until`' command would appear to step back to the beginning of the loop when it advances to this expression. However, it has not really done so, not in terms of the actual machine code.

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

`until` *location*
Continue running the program until either the specified location is reached, or the current (innermost) stack frame returns. This form of the command uses breakpoints, and hence is quicker than '`until`' without an argument.

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

It is often useful to do '`display/i $pc`' when stepping by machine instructions. This will cause the next instruction to be executed to be displayed automatically at each stop. See Section 8.6 [Auto Display], page 40.

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

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

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

A typical technique for using stepping is to put a breakpoint (see Section 5.2 [Breakpoints], page 18) at the beginning of the function or the section of the program in which a problem is believed to lie, and then step through the suspect area, examining the variables that are interesting, until the problem happens.

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

6 Examining the Stack

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

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

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

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

When the program stops, GDB automatically selects the currently executing frame and describes it briefly as the '`frame`' command does (see Section 6.4 [Frame Info], page 29).

6.1 Stack Frames

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

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

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

GDB assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your program; they are to give you a way of talking about stack frames in GDB commands.

Many GDB commands refer implicitly to one stack frame. GDB records a stack frame that is called the *selected* stack frame; you can select any frame using one set of GDB commands, and then other commands will operate on that frame. When your program stops, GDB automatically selects the innermost frame.

Some functions can be compiled to run without a frame reserved for them on the stack. This is occasionally done with heavily used library functions to save the frame setup time. GDB has limited facilities for dealing with these function invocations; if the innermost

function invocation has no stack frame, GDB will give it a virtual stack frame of 0 and correctly allow tracing of the function call chain. Results are undefined if a function invocation besides the innermost one is frameless.

6.2 Backtraces

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

`backtrace`
`bt` Print a backtrace of the entire stack: one line per frame for all frames in the stack.

You can stop the backtrace at any time by typing the system interrupt character, normally *Control-C*.

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

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

The names '`where`' and '`info stack`' are additional aliases for '`backtrace`'.

Every line in the backtrace shows the frame number, the function name and the program counter value.

If the function is in a source file whose symbol table data has been fully read, the backtrace shows the source file name and line number, as well as the arguments to the function. (The program counter value is omitted if it is at the beginning of the code for that line number.)

If the source file's symbol data has not been fully read, just scanned, this extra information is replaced with an ellipsis. You can force the symbol data for that frame's source file to be read by selecting the frame. (See Section 6.3 [Selection], page 28).

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

```
#0  rtx_equal_p (x=(rtx) 0x8e58c, y=(rtx) 0x1086c4) (/gp/rms/cc/rtlanal.c line 337)
#1  0x246b0 in expand_call (...) (...)
#2  0x21cfc in expand_expr (...) (...)
(More stack frames follow...)
```

The functions `expand_call` and `expand_expr` are in a file whose symbol details have not been fully read. Full detail is available for the function `rtx_equal_p`, which is in the file `rtlanal.c`. Its arguments, named `x` and `y`, are shown with their typed values.

6.3 Selecting a Frame

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

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

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

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

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

All of these commands end by printing some information on the frame that has been selected: the frame number, the function name, the arguments, the source file and line number of execution in that frame, and the text of that source line. For example:

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

After such a printout, the '`list`' command with no arguments will print ten lines centered on the point of execution in the frame. See Section 7.1 [List], page 31.

6.4 Information on a Frame

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

`frame` This command prints a brief description of the selected stack frame. It can be abbreviated '`f`'. With an argument, this command is used to select a stack frame; with no argument, it does not change which frame is selected, but still prints the same information.

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

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

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

`info locals`

Print the local variables of the selected frame, each on a separate line. These are all variables declared static or automatic within all program blocks that execution in this frame is currently inside of.

7 Examining Source Files

GDB knows which source files your program was compiled from, and can print parts of their text. When your program stops, GDB spontaneously prints the line it stopped in. Likewise, when you select a stack frame (see Section 6.3 [Selection], page 28), GDB prints the line which execution in that frame has stopped in. You can also print parts of source files by explicit command.

7.1 Printing Source Lines

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

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

`list` *linenum*
: Print ten lines centered around line number *linenum* in the current source file.

`list` *function*
: Print ten lines centered around the beginning of function *function*.

`list`
: Print ten more lines. If the last lines printed were printed with a '`list`' command, this prints ten 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 [Stack], page 27), this prints ten lines centered around that line.

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

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 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 ten lines centered around the line specified by *linespec*.

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

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

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

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

`list -`
: Print ten 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.

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

function Specifies the line of the open-brace that begins the body of the function *function*.

filename`:`*function*
Specifies the line of the open-brace that begins the body of the function *function* in the file *filename*. The file name is needed with a function name only for disambiguation of identically named functions in different source files.

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

One other command is used to map source lines to program addresses.

`info line` *linenum*
Print the starting and ending addresses of the compiled code for source line *linenum*.

The default examine 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 8.5.1 [Memory], page 38). Also, this address is saved as the value of the convenience variable `$_` (see Section 8.8 [Convenience Vars], page 42).

7.2 Searching Source Files

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

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 abbreviate the command name as '`fo`'.

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 with as little as '`rev`'.

7.3 Specifying Source Directories

Executable programs do not record the directories of the source files from which they were compiled, just the names. GDB remembers a list of directories to search for source files; this is called the *source path*. Each time GDB wants a source file, it tries all the directories in the list, in the order they are present in the list, until it finds a file with the desired name.

Note that the executable search path is *not* used for this purpose. Neither is the current working directory, unless it happens to be in the source path.

When you start GDB, its source path contains just the current working directory. To add other directories, use the '`directory`' command.

`directory` *`dirnames...`*

Add directory *dirname* to the end of the source path. Several directory names may be given to this command, separated by whitespace or '`:`'.

`directory`

Reset the source path to just the current working directory of GDB. This requires confirmation.

Since this command deletes directories from the search path, it may change the directory in which a previously read source file will be discovered. To make this work correctly, this command also clears out the tables GDB maintains about the source files it has already found.

`info directories`

Print the source path: show which directories it contains.

Because the '`directory`' command adds to the end of the source path, it does not affect any file that GDB has already found. If the source path contains directories that you do not want, and these directories contain misleading files with names matching your source files, the way to correct the situation is as follows:

1. Choose the directory you want at the beginning of the source path. Use the '`cd`' command to make that the current working directory.
2. Use '`directory`' with no argument to reset the source path to just that directory.
3. Use '`directory`' with suitable arguments to add any other directories you want in the source path.

8 Examining Data

The usual way to examine data in your program is with the '`print`' command (abbreviated '`p`'). It evaluates and prints the value of any valid expression of the language the program is written in (for now, C). You type

```
print exp
```

where *exp* is any valid expression, and the value of *exp* is printed in a format appropriate to its data type.

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.

8.1 Expressions

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

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

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

`@` '`@`' is a binary operator for treating parts of memory as arrays. See Section 8.3 [Arrays], page 36, for more information.

`::` '`::`' allows you to specify a variable in terms of the file or function it is defined in. See Section 8.2 [Variables], page 35.

`{`*type*`}` *addr*
: Refers to an object of type *type* stored at address *addr* in memory. *addr* may be any expression whose value is an integer or pointer (but parentheses are required around nonunary operators, just as in a cast). This construct is allowed regardless of what kind of data is officially 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 [Selection], page 28); they must either be global (or static) or be visible according to the scope rules of the programming language from the point of execution in that frame. This means that in the function

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

```
  }
}
```

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

As a special 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 (if they are in different source files). In such a case, it is not defined which one you will get. If you wish, you can specify any one of them using the colon-colon construct:

`block``::`*`variable`*

Here *block* is the name of the source file whose variable you want.

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.

This can be done by constructing an *artificial array* with the binary operator '`@`'. The left operand of '`@`' should be the first element of the desired array, as an individual object. The right operand should be the length of the array. The result is an array value whose elements are all of the type of the left argument. The first element is actually the left argument; the second element comes from bytes of memory immediately following those that hold the first element, and so on. 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. (It would probably appear in an expression via the value history, after you had printed it out.)

8.4 Format options

GDB provides a few ways to control how arrays and structures are printed.

`info format`
: Display the current settings for the format options.

`set array-max` *`number-of-elements`*
: If GDB is printing a large array, it will stop printing after it has printed the number of elements set by the '`set array-max`' command. This limit also applies to the display of strings.

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

```
$1 = {
```

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

`set prettyprint off`

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

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

This is the default format.

`set unionprint on`

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

`set unionprint off`

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

For example, given the declarations

```
typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpiller, Cocoon, Butterfly} Bug_forms;

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

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

with '`set unionprint on`' in effect '`p foo`' would print

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

and with '`set unionprint off`' in effect it would print

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

8.5 Output formats

GDB normally prints all values according to their data types. 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 an instruction. These things can be done with *output formats*.

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.

'a' Print as an address, both absolute in hex and then relative to a symbol defined as an address below it.

'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 8.9 [Registers], page 43), 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.

8.5.1 Examining Memory

The command 'x' (for 'examine') can be used to examine memory without reference to the program's data types. The format in which you wish to examine memory is instead explicitly specified. The allowable formats are a superset of the formats described in the previous section.

'x' is followed by a slash and an output format specification, followed by an expression for an address. The expression need not have a pointer value (though it may); it is used as an integer, as the address of a byte of memory. See Section 8.1 [Expressions], page 35, for more information on expressions. For example, 'x/4xw $sp' prints the four words of memory above the stack pointer in hexadecimal.

The output format in this case specifies both how big a unit of memory to examine and how to print the contents of that unit. It is done with one or two of the following letters:

These letters specify just the size of unit to examine:

'b' Examine individual bytes.

'h' Examine halfwords (two bytes each).

'w' Examine words (four bytes each).

Many assemblers and cpu designers still use 'word' for a 16-bit quantity, as a holdover from specific predecessor machines of the 1970's that really did use two-byte words. But more generally the term 'word' has always referred to the size of quantity that a machine normally operates on and stores in its registers. This is 32 bits for all the machines that GDB runs on.

'g' Examine giant words (8 bytes).

These letters specify just the way to print the contents:

‘`x`’ Print as integers in unsigned hexadecimal.

‘`d`’ Print as integers in signed decimal.

‘`u`’ Print as integers in unsigned decimal.

‘`o`’ Print as integers in unsigned octal.

‘`a`’ Print as an address, both absolute in hex and then relative to a symbol defined as an address below it.

‘`c`’ Print as character constants.

‘`f`’ Print as floating point. This works only with sizes ‘`w`’ and ‘`g`’.

‘`s`’ Print a null-terminated string of characters. The specified unit size is ignored; instead, the unit is however many bytes it takes to reach a null character (including the null character).

‘`i`’ Print a machine instruction in assembler syntax (or nearly). The specified unit size is ignored; the number of bytes in an instruction varies depending on the type of machine, the opcode and the addressing modes used.

If either the manner of printing or the size of unit fails to be specified, the default is to use the same one that was used last. If you don't want to use any letters after the slash, you can omit the slash as well.

You can also omit the address to examine. Then the address used is just after the last unit examined. This is why string and instruction formats actually compute a unit-size based on the data: so that the next string or instruction examined will start in the right place. The ‘`print`’ command sometimes sets the default address for the ‘`x`’ command; when the value printed resides in memory, the default is set to examine the same location. ‘`info line`’ also sets the default for ‘`x`’, to the address of the start of the machine code for the specified line and ‘`info breakpoints`’ sets it to the address of the last breakpoint listed.

When you use `RET` to repeat an ‘`x`’ command, it does not repeat exactly the same: the address specified previously (if any) is ignored, so that the repeated command examines the successive locations in memory rather than the same ones.

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

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

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

```
x/10
```

in which the format and address are allowed to default.

The addresses and contents printed by the ‘`x`’ command are not put 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.

The specialized command '`disassemble`' is also provided to dump 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; the function surrounding this value will be dumped. Two arguments specify a range of addresss (first inclusive, second exclusive) to be dumped.

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 will print its value each time the program stops. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

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

showing item numbers, expressions and their current values.

If the expression refers to local variables, then it does not make sense outside the lexical context for which it was set up. Such an expression is printed only when execution is inside that lexical context. For example, if you give the command '`display name`' while inside a function with an argument `name`, then this argument will be displayed whenever the program stops inside that function, but not when it stops elsewhere (since this argument doesn't exist elsewhere).

`display` *exp*
: Add the expression *exp* to the list of expressions to display each time the program stops. See Section 8.1 [Expressions], page 35.

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

`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 the program stops. Examining means in effect doing '`x/`*fmt addr*'. See Section 8.5.1 [Memory], page 38.

`undisplay` *dnums...*
`delete display` *dnums...*
: Remove item numbers *dnums* from the list of expressions to display.

`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 reenabled 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 the 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.

8.7 Value History

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

The values printed are given *history numbers* for you to refer to them by. These are successive integers starting with 1. '`print`' shows you the history number assigned to a value by printing '`$`*num* `= `' before the value; here *num* is the history number.

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

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

```
p *$
```

If you have a chain of structures where the component '`next`' points to the next one, you can print the contents of the next one with this:

```
p *$.next
```

It might be useful to repeat this command many times by typing `RET`.

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

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

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

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

`info values` *n*
: Print ten history values centered on history item number *n*.

`info values +`
: Print ten history values just after the values last printed.

8.8 Convenience Variables

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

Convenience variables have names starting with '`$`'. Any name starting with '`$`' can be used for a convenience variable, unless it is one of the predefined set of register names (see Section 8.9 [Registers], page 43).

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

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

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

Using a convenience variable for the first time creates it; but its value is `void` until you assign a new value. You can alter the value with another assignment at any time.

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

`info convenience`
: Print a list of convenience variables used so far, and their values. Abbreviated '`i con`'.

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

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

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

`$_`
: The variable `$_` is automatically set by the '`x`' command to the last address examined (see Section 8.5.1 [Memory], page 38). Other commands which provide a default address for '`x`' to examine also set `$_` to that address; these commands include '`info line`' and '`info breakpoint`'.

`$__`
: The variable `$__` is automatically set by the '`x`' command to the value found in the last address examined.

8.9 Registers

Machine register contents can be referred to in expressions as variables with names starting with '`$`'. The names of registers are different for each machine; use '`info registers`' to see the names used on your machine. The names `$pc` and `$sp` are used on all machines for the program counter register and the stack pointer. Often `$fp` is used for a register that contains a pointer to the current stack frame, and `$ps` is used for a register that contains the processor status. These standard register names may be available on your machine even though the `info registers` command displays them with a different name. For example, on the SPARC, `info registers` displays the processor status register as `$psr` but you can also refer to it as `$ps`.

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

Some registers have distinct "raw" and "virtual" data formats. This means that the data format in which the register contents are saved by the operating system is not the same one that your program normally sees. For example, the registers of the 68881 floating point coprocessor are always saved in "extended" format, but all C programs expect to work with "double" format. In such cases, GDB normally works with the virtual format only (the format that makes sense for your program), but the '`info registers`' command prints the data in both formats.

Register values are relative to the selected stack frame (see Section 6.3 [Selection], page 28). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the real contents of all registers, you must select the innermost frame (with '`frame 0`').

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

`info registers`
: Print the names and relativized values of all registers.

`info registers` *regname*
: Print the relativized value of register *regname*. *regname* may be any register name valid on the machine you are using, with or without the initial '`$`'.

8.9.1 Examples

You could print the program counter in hex with

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add four to the stack pointer with

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

The last is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected. Setting `$sp` is not allowed when other stack frames are selected.

9 Examining the Symbol Table

The commands described in this section allow you to make inquiries for information about the symbols (names of variables, functions and types) defined in your program. This information is found by GDB in the symbol table loaded by the '`symbol-file`' command; it is inherent in the text of your program and does not change as the program executes.

`whatis` *exp*
: Print the data type of expression *exp*. *exp* is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place. See Section 8.1 [Expressions], page 35.

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

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

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

`ptype` *typename*
: Print a description of data type *typename*. *typename* may be the name of a type, or for C code it may have the form '`struct` *struct-tag*', '`union` *union-tag*' or '`enum` *enum-tag*'.

`info sources`
: Print the names of all source files in the program for which there is debugging information.

`info functions`
: Print the names and data types of all defined functions.

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

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

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

`info types`
: Print all data types that are defined in the program.

`info types` *regexp*
: Print all data types that are defined in the program whose names contain a match for regular expression *regexp*.

`info methods`
`info methods` *regexp*
: The '`info-methods`' command permits the user to examine all defined methods within C++ program, or (with the *regexp* argument) a specific set of methods found in the various C++ classes. Many C++ classes provide a large number of methods. Thus, the output from the '`ptype`' command can be overwhelming and hard to use. The '`info-methods`' command filters the methods, printing only those which match the regular-expression *regexp*.

`printsyms` *filename*
: Write a complete dump of the debugger's symbol data into the file *filename*.

10 Altering Execution

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

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

10.1 Assignment to Variables

To alter the value of a variable, evaluate an assignment expression. See Section 8.1 [Expressions], page 35. For example,

```
print x=4
```

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

All the assignment operators of C are supported, including the incrementation operators '`++`' and '`--`', and combining assignments such as '`+=`' and '`<<=`'.

If you are not interested in seeing the value of the assignment, use the '`set`' command instead of the '`print`' command. '`set`' is really the same as '`print`' except that the expression's value is not printed and is not put in the value history (see Section 8.7 [Value History], page 41). The expression is evaluated only for side effects.

Note that if the beginning of the argument string of the '`set`' command appears identical to a '`set`' subcommand, it may be necessary to use the '`set variable`' command. This command is identical to '`set`' except for its lack of subcommands.

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

To store values into arbitrary places in memory, use the '`{...}`' construct to generate a value of specified type at a specified address (see Section 8.1 [Expressions], page 35). For example, `{int}0x83040` would refer to memory location 0x83040 as an integer (which implies a certain size and representation in memory), and

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

would store the value 4 into that memory location.

10.2 Continuing at a Different Address

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

`jump` *`linenum`*

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

The '`jump`' command does not change the current stack frame, or the stack pointer, or the contents of any memory location or any register other than the program counter. If line *linenum* is in a different function from the one currently executing, the results may be bizarre if the two functions expect different patterns of arguments or of local variables. For this reason, the '`jump`' command requests confirmation if the specified line is not in the function currently executing. However, even bizarre results are predictable based on careful study of the machine-language code of the program.

`jump *`*address*

Resume execution at the instruction at address *address*.

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

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

causes the next '`cont`' command or stepping command to execute at address 0x485, rather than at the address where the program stopped. See Section 5.4 [Stepping], page 24.

The most common occasion to use the '`jump`' command is when you have stepped across a function call with `next`, and found that the return value is incorrect. If all the relevant data appeared correct before the function call, the error is probably in the function that just returned.

In general, your next step would now be to rerun the program and execute up to this function call, and then step into it to see where it goes astray. But this may be time consuming. If the function did not have significant side effects, you could get the same information by resuming execution just before the function call and stepping through it. To do this, first put a breakpoint on that function; then, use the '`jump`' command to continue on the line with the function call.

10.3 Giving the Program a Signal

`signal` *signalnum*

Resume execution where the program stopped, but give it immediately the signal number *signalnum*.

Alternatively, if *signalnum* is zero, continue execution without giving a signal. This is useful when the program stopped on account of a signal and would ordinary see the signal when resumed with the '`cont`' command; '`signal 0`' causes it to resume without a signal.

10.4 Returning from a Function

You can cancel execution of a function call with the '`return`' command. This command has the effect of discarding the selected stack frame (and all frames within it), so that control moves to the caller of that function. You can think of this as making the discarded frame return prematurely.

First select the stack frame that you wish to return from (see Section 6.3 [Selection], page 28). Then type the '`return`' command. If you wish to specify the value to be returned, give that as an argument.

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

The '`return`' command does not resume execution; it leaves the program stopped in the state that would exist if the function had just returned. Contrast this with the '`finish`' command (see Section 5.4 [Stepping], page 24), which resumes execution until the selected stack frame returns *naturally*.

11 Canned Sequences of Commands

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

11.1 User-Defined Commands

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

`define` *commandname*
: Define a command named *commandname*. If there is already a command by that name, you are asked to confirm that you want to redefine it.

  The definition of the command is made up of other GDB command lines, which are given following the '`define`' command. The end of these commands is marked by a line containing '`end`'.

`document` *commandname*
: Give documentation to the user-defined command *commandname*. The command *commandname* must already be defined. This command reads lines of documentation just as '`define`' reads the lines of the command definition, ending with '`end`'. After the '`document`' command is finished, '`help`' on command *commandname* will print the documentation you have specified.

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

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

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

11.2 Command Files

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

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

`source` *filename*
: Execute the command file *filename*.

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

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

11.3 Commands for Controlled Output

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

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

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

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

produces the same output as

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

`output` *expression*
Print the value of *expression* and nothing but that value: no newlines, no '`$`*nn* `=` '. The value is not entered in the value history either. See Section 8.1 [Expressions], page 35, for more information on expressions.

`output/`*fmt* *expression*
Print the value of *expression* in format *fmt*. See Section 8.5 [Output formats], page 37, for more information.

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

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

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

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

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

12 Options and Arguments for GDB

When you invoke GDB, you can specify arguments telling it what files to operate on and what other things to do.

12.1 Mode Options

'`-nx`' Do not execute commands from the init files `.gdbinit`. Normally, the commands in these files are executed after all the command options and arguments have been processed. See Section 11.2 [Command Files], page 51.

'`-q`' "Quiet". Do not print the usual introductory messages.

'`-batch`' Run in batch mode. Exit with code 0 after processing all the command files specified with '`-x`' (and `.gdbinit`, if not inhibited). Exit with nonzero status if an error occurs in executing the GDB commands in the command files.

'`-fullname`'
This option is used when Emacs runs GDB as a subprocess. It tells GDB to output the full file name and line number in a standard, recognizable fashion each time a stack frame is displayed (which includes each time the program stops). This recognizable format looks like two '`\032`' characters, followed by the file name, line number and character position separated by colons, and a newline. The Emacs-to-GDB interface program uses the two '`\032`' characters as a signal to display the source code for the frame.

12.2 File-specifying Options

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

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

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

'`-se` *file*' Read symbol table from file *file* and use it as the executable file.

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

'`-x` *file*' Execute GDB commands from file *file*.

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

12.3 Other Arguments

If there are arguments to GDB that are not options or associated with options, the first one specifies the symbol table and executable file name (as if it were preceded by '`-se`') and the second one specifies a core dump file name (as if it were preceded by '`-c`').

13 Using GDB under GNU Emacs

A special interface allows you to use GNU Emacs to view (and edit) the source files for the program you are debugging with GDB.

To use this interface, use the command *M-x gdb* in Emacs. Give the executable file you want to debug as an argument. This command starts GDB as a subprocess of Emacs, with input and output through a newly created Emacs buffer.

Using GDB under Emacs is just like using GDB normally except for two things:

- All "terminal" input and output goes through the Emacs buffer. This applies both to GDB commands and their output, and to the input and output done by the program you are debugging.

  This is useful because it means that you can copy the text of previous commands and input them again; you can even use parts of the output in this way.

  All the facilities of Emacs's Shell mode are available for this purpose.
- GDB displays source code through Emacs. Each time GDB displays a stack frame, Emacs automatically finds the source file for that frame and puts an arrow (‘=>’) at the left margin of the current line.

  Explicit GDB ‘`list`’ or search commands still produce output as usual, but you probably will have no reason to use them.

In the GDB I/O buffer, you can use these special Emacs commands:

M-s	Execute to another source line, like the GDB ‘`step`’ command.
M-n	Execute to next source line in this function, skipping all function calls, like the GDB ‘`next`’ command.
M-i	Execute one instruction, like the GDB ‘`stepi`’ command.
C-c C-f	Execute until exit from the selected stack frame, like the GDB ‘`finish`’ command.
C-c C-c	Continue execution of the program, like the GDB ‘`cont`’ command.

In any source file, the Emacs command *C-x SPC* (`gdb-break`) tells GDB to set a breakpoint on the source line point is on.

The source files displayed in Emacs are in ordinary Emacs buffers which are visiting the source files in the usual way. You can edit the files with these buffers if you wish; but keep in mind that GDB communicates with Emacs in terms of line numbers. If you add or delete lines from the text, the line numbers that GDB knows will cease to correspond properly to the code.

14 Remote Kernel Debugging

GDB has a special facility for debugging a remote machine via a serial connection. This can be used for kernel debugging.

The program to be debugged on the remote machine needs to contain a debugging device driver which talks to GDB over the serial line using the protocol described below. The same version of GDB that is used ordinarily can be used for this.

For details of the communication protocol, see the comments in the GDB source file `remote.c`.

14.1 Commands for Remote Debugging

To start remote debugging, first run GDB and specify as an executable file the program that is running in the remote machine. This tells GDB how to find the program's symbols and the contents of its pure text. Then establish communication using the '`attach`' command with a device name rather than a pid as an argument. For example:

```
attach /dev/ttyd
```

if the serial line is connected to the device named `/dev/ttyd`. This will stop the remote machine if it is not already stopped.

Now you can use all the usual commands to examine and change data and to step and continue the remote program.

To resume the remote program and stop debugging it, use the '`detach`' command.

Command Index

(Index is nonexistent)

Concept Index

$

$ 41
$$ 41
$_ 18, 32, 39
$__ 39

.

.gdbinit 51

A

abbreviation 7
add-file 10
arguments (to your program) 13
artificial array 36
assignment 47
attach 15
automatic display 40

B

backtrace 28
break 18
breakpoint commands 22
breakpoint conditions 21
breakpoints 18
bt 28

C

call stack 27
cd 14
clear 20
clearing breakpoint 19
command files 51
condition 21
conditional breakpoints 21
cont 23
controlling terminal 14
convenience variables 42
core dump file 9
core-file 10

D

define 51
delete 20
delete display 40
delete environment 14
deleting breakpoints 19
detach 15
directories for source files 32
directory 33
disable 20
disable breakpoints 20
disable display 41
disabled breakpoints 20
disassemble 40
display 40
display of expressions 40
document 51
down 29
dynamic linking 10

E

echo 52
enable 20
enable breakpoints 20
enable display 41
enabled breakpoints 20
environment (of your program) 13
examining data 35
examining memory 38
exec-file 9
executable file 9
exiting GDB 7
expressions 35

F

fatal signals 17
finish 24
format options 36
formatted output 37
forward-search 32
frame 27, 29
frame number 27
frame pointer 27
frameless execution 27

H

handle 17
handling signals 17
history number 41

I

ignore 21
ignore count (of breakpoint) 21
info address 45
info args 29
info break 18
info convenience 42
info directories 33
info display 41
info environment 14
info files 10
info format 36
info frame 29
info functions 45
info line 32
info locals 30
info methods 46
info registers 43
info signal 17
info sources 45
info stack 28
info types 45
info values 41
info variables 45
init file 51
initial frame 27
innermost frame 27

J

jump 47

K

kill 15

L

linespec 31
list 31

N

next 24
nexti 25
ni 25

O

outermost frame 27
output 52
output formats 37

P

pauses in output 7
print 35
printf 52
printing data 35
printsyms 46
prompt 7
ptype 45
pwd 14

Q

quit 7

R

redirection 14
registers 43
repeating commands 7
return 48
returning from a function 48
reverse-search 32
run 13
running 13

S

screen size 7
searching 32
selected frame 27
set 47
set args 13
set array-max 36
set environment 14
set prettyprint 36
set prompt 7
set screensize 7
set unionprint 37
set variable 47
set verbose 8
setting variables 47
si 25
signal 48
signals 17
silent 22
source 51
source path 32
stack frame 27
step 24
stepi 25
stepping 24
symbol table 9
symbol-file 9

T

tbreak 19
tty 15

U

undisplay 40
unset environment 14
until 25
up 29
user-defined command 51

V

value history 41

W

whatis 45
where 28
word 38
working directory (of your program) 14

X

x 38

Table of Contents

GNU GENERAL PUBLIC LICENSE **1**
Preamble 1
TERMS AND CONDITIONS 1
Appendix: How to Apply These Terms to Your New Programs 5

1 GDB Input and Output Conventions **7**

2 Specifying GDB's Files **9**
2.1 Specifying Files with Arguments 9
2.2 Specifying Files with Commands 9

3 Compiling Your Program for Debugging **11**

4 Running Your Program Under GDB **13**
4.1 Your Program's Arguments 13
4.2 Your Program's Environment 13
4.3 Your Program's Working Directory 14
4.4 Your Program's Input and Output 14
4.5 Debugging an Already-Running Process 15
4.6 Killing the Child Process 15

5 Stopping and Continuing **17**
5.1 Signals 17
5.2 Breakpoints 18
5.2.1 Setting Breakpoints 18
5.2.2 Deleting Breakpoints 19
5.2.3 Disabling Breakpoints 20
5.2.4 Break Conditions 21
5.2.5 Commands Executed on Breaking 22
5.2.6 "Cannot Insert Breakpoints" Error 23
5.3 Continuing 23
5.4 Stepping 24

6 Examining the Stack **27**
6.1 Stack Frames 27
6.2 Backtraces 28
6.3 Selecting a Frame 28
6.4 Information on a Frame 29

7 Examining Source Files 31
7.1 Printing Source Lines 31
7.2 Searching Source Files 32
7.3 Specifying Source Directories 32

8 Examining Data 35
8.1 Expressions 35
8.2 Program Variables 35
8.3 Artificial Arrays 36
8.4 Format options 36
8.5 Output formats 37
8.5.1 Examining Memory 38
8.6 Automatic Display 40
8.7 Value History 41
8.8 Convenience Variables 42
8.9 Registers 43
8.9.1 Examples 43

9 Examining the Symbol Table 45

10 Altering Execution 47
10.1 Assignment to Variables 47
10.2 Continuing at a Different Address 47
10.3 Giving the Program a Signal 48
10.4 Returning from a Function 48

11 Canned Sequences of Commands 51
11.1 User-Defined Commands 51
11.2 Command Files 51
11.3 Commands for Controlled Output 52

12 Options and Arguments for GDB 53
12.1 Mode Options 53
12.2 File-specifying Options 53
12.3 Other Arguments 53

13 Using GDB under GNU Emacs 55

14 Remote Kernel Debugging 57
14.1 Commands for Remote Debugging 57

Command Index 59

Concept Index **61**