Debugging Ada Programs

Using the XGC Ada Debugger

www.xgc.com

Debugging Ada Programs

Using the XGC Ada Debugger

Order Number: XGC-ADA-1.8-GDB-110803

XGC Technology

London
UK
Web: `<www.xgc.com>`

Debugging Ada Programs: Using the XGC Ada Debugger

by Free Software Foundation and XGC Technology

Publication date August 2011

Acknowledgments

This guide is based on documentation distributed by the Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

License

Contents

Examples

About this Guide

This guide contains detailed *target independent* information about the XGC Ada debugger, including all the command line options. It includes the text of the GNU debugger (GDB) user manual, with examples for Ada 95.

When using the examples, the parameter `prefix` should be replaced with the prefix that is applicable for your product. You will find this information in the *Getting Started* manual.

1. Audience

This guide is written for the experienced programmer who is already familiar with the Ada 95 and C programming languages and with embedded systems programming in general. We assume some knowledge of the target computer architecture.

2. Related Documents

Getting Started with XGC Ada describes how to to prepare and run a simple program, and contains target dependent information that supplements the other user manuals.

The XGC Ada Reference Manual Supplement contains information required by the Ada Reference Manual.

The XGC Ada User Guide describes the compiler and Ada utilities.

The XGC Utilities describes the assembler, linker and object code utilities.

The *XGC Libraries* documents the library functions available with all XGC compilers.

3. Reader's Comments

We welcome any comments and suggestions you have on this and other XGC user manuals.

You can send your comments in the following ways:

- Internet electronic mail: readers_comments@xgc.com

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of the software that you are using.

Technical support enquiries should be directed to the XGC web site [http://www.xgc.com/] or by email to support@xgc.com/.

4. Documentation Conventions

This guide uses the following typographic conventions:

`%, $`
A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bash shell.

`#`
A number sign represents the superuser prompt.

`$` **`vi hello.c`**

Boldface type in interactive examples indicates typed user input.

`file`

Italic or slanted type indicates variable values, place-holders, and function argument names.

[|], { | }

In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.

...

In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated.

cat(1)

A cross-reference to a reference page includes the appropriate section number in parentheses. For example, cat(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.

Mb/s

This symbol indicates megabits per second.

MB/s

This symbol indicates megabytes per second.

Ctrl+**x**

This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows. In examples, this key combination is printed in bold type (for example, **Ctrl**+**C**).

Chapter 1

A Sample Debug Session

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

The benchmark program *Whetstone* is frequently used to measure the performance of floating-point operations. We run Whetstone on the target to compare its performance with other computers. The workings of Whetstone are a bit of a mystery, but by using the debugger we can get an insight.

The first step is to compile the Whetstone source with the debug option switched on. There is no need to compile with minimal optimization since the debugger is able to cope with most optimizations.

```
$ prefix-gcc -g whetstone.adb -o whetstone
```

Note that `prefix` should be replaced with the prefix for your product.

The code we wish to look at starts on line 306, just after the array e1 has been set up. Here is part of the Ada source file showing line 306 and some of the surrounding lines.

```
300
301            -- Module 2: computations with array elements
302            e1 (1) := 1.0;
303            e1 (2) := -1.0;
304            e1 (3) := -1.0;
305            e1 (4) := -1.0;
306            for i in 1 .. n2 loop
307               e1 (1) := (e1 (1) + e1 (2) + e1 (3) - e1 (4)) * t;
308               e1 (2) := (e1 (1) + e1 (2) - e1 (3) + e1 (4)) * t;
```

We start the debugger using the following command. You may use the **-q** option to suppress the banner:

```
$ prefix-gdb whetstone
XGC target-ada Version 1.5 (debugger)
Copyright (c) 1996, 2001, XGC Technology.
Based on gdb version 4.17.gnat.3.11
Copyright (c) 1998 Free Software Foundation...
(gdb)
```

The easiest way to get to line 306 is to set a breakpoint on that line, then use the run command. Breakpoints are set using the **break** command. We can check what breakpoints are set using the **info** command.

```
(gdb) break whetstone.adb:306
Breakpoint 1 at 0xee2: file whetstone.adb, line 306.
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00000ee2 in whetstone at whetstone.adb:306
(gdb)
```

So far we have been working with the *exec* target. This is the executable file `whetstone` cited on the comand line that invoked the debugger. Using just this file, and no target at all, we can look at the values of symbols, inspect the source code, and the generated code, and check the values of static variables.

In order to run the program we must switch to a real or simulated target. The debugger supports both. The simulator target is called `sim` and a real target is called `remote`. The debugger will automatically switch to the simulator and load the program into the simulator if we enter the run command at this point.

```
(gdb) run
Starting program: .../examples/whetstone
Connected to the simulator.
Loading sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .init         00000408  00000000  00000000  00001000  2**1
                  CONTENTS, ALLOC, LOAD, CODE
  1 .text         00001890  00000408  00000408  00001408  2**1
                  CONTENTS, ALLOC, LOAD, CODE
  2 .rdata        000003ce  00001c98  00001c98  00002c98  2**1
                  CONTENTS, ALLOC, LOAD, READONLY
  3 .data         0000038a  00010000  00002066  00004000  2**1
                  CONTENTS, ALLOC, LOAD, DATA
Start address 0x0
Transfer rate: 73600 bits in <1 sec.
Whetstone: Floating point benchmark

Breakpoint 1, whetstone () at whetstone.adb:306
306           for i in 1 .. n2 loop
(gdb)
```

What we want to do now is see how the value of e1 changes as we go round the loop. We can print the initial value using the print command. Note that arrays (and structures) may be printed with a single command.

```
(gdb) print e1(1)
$1 = 1
(gdb) print e1(2)
$2 = -1
(gdb) print e1
$2 = (1 => 1, 2 => -1, 3 => -1, 4 => -1)
(gdb)
```

We can then step through the program one line at a time using the next command. Like many other commands, the next command repeats when we press **Enter**. Therefore, after the first next command, we just hit **Enter**.

```
(gdb) next
307              e1 (1) := (e1 (1) + e1 (2) + e1 (3) - e1 (4)) * t;
(gdb) enter
308              e1 (2) := (e1 (1) + e1 (2) - e1 (3) + e1 (4)) * t;
(gdb) enter
309              e1 (3) := (e1 (1) - e1 (2) + e1 (3) + e1 (4)) * t;
```

```
(gdb) enter
310                 e1 (4) := (-e1 (1) + e1 (2) + e1 (3) + e1 (4)) * t;
(gdb)
```

Let's check the value of e1. This time we'll use the abbreviation.

```
(gdb) p e1
$3 = (1 => 0, 2 => -0.499975026, 3 => -0.749975085, 4 => -1)
```

If we need to check the value of e1 each time round the loop, it is tedious to have to step through each line then type the print command at the end of the loop. Instead we can place a breakpoint at the end of the loop and use the continue command to execute to the end of the loop. Furthermore, we can use the display command to print the value of e1 each time the program stops.

```
(gdb) br
Breakpoint 2 at 0xf2e: file whetstone.adb, line 310.
(gdb) display e1
1: e1 = (1 => 0, 2 => -0.499975026, 3 => -0.749975085, 4 => -1)
```

Use the continue command to run the program to the next breakpoint. Then press **Enter** to repeat the continue command. Note that **continue** may be abbreviated to **c**.

```
(gdb) c
Continuing.

Breakpoint 2, whetstone () at whetstone.adb:310
310                 e1 (4) := (-e1 (1) + e1 (2) + e1 (3) + e1 (4)) * t;
1: e1 = (1 => -0.0625123829, 2 => -0.468692422, 3 => -0.734320521, 4 => -1.12
(gdb) Enter
Continuing.

Breakpoint 2, whetstone () at whetstone.adb:310
310                 e1 (4) := (-e1 (1) + e1 (2) + e1 (3) + e1 (4)) * t;
1: e1 = (1 => -0.0664326251, 2 => -0.466705739, 3 => -0.733313918, 4 => -1.13
(gdb) Enter
```

To finish the debugging session use the quit command. You may abbreviate **quit** to **q**.

```
(gdb) q
The program is running. Quit anyway (and kill it)? (y or n) y
$
```

Chapter 2 *Getting In and Out of the Debugger*

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

- type the command ***prefix*-gdb** to start the debugger.
- type **quit** or **Ctrl**+**d** to exit.

2.1. Invoking the Debugger

Invoke the debugger by entering the command ***prefix*-gdb**. Once started, the debugger reads commands from the terminal until you tell it to exit.

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

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

```
$ prefix-gdb program
```

You can run the debugger without printing the front material, which describes warranty, by specifying **-silent**:

```
$ prefix-gdb -silent
```

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

Type

```
$ prefix-gdb -help
```

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

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

2.1.1. Choosing Files

When the debugger starts, it reads any argument other than options as specifying an executable file. This is the same as if the argument was specified by the **-se** option.

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

-symbols *file* , -s *file*
: Read symbol table from file *file*.

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

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

-command `file` , -x `file`
Execute debug commands from file `file`. See Section 15.3, “Command Files” [126].

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

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

2.1.2. Choosing Modes

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

-nx , -n
Do not execute commands from any initialization files (normally called `.gdbinit`). Normally, the commands in these files are executed after all the command options and arguments have been processed. See Section 15.3, “Command Files” [126].

-quiet , -q
Do not print the introductory and copyright messages. These messages are also suppressed in batch mode.

-batch
Run in batch mode. Exit with status **0** after processing all the command files specified with **-x** (and all commands from initialization files, if not inhibited with **-n**). Exit with nonzero status if an error occurs in executing the debug commands in the command files.

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

```
Program exited normally.
```

(which is ordinarily issued whenever a program running under the debugger control terminates) is not issued when running in batch mode.

-cd ***`directory`***
: Run the debugger using *`directory`* as its working directory, instead of the current directory.

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

-b ***`bps`***
: Set the line speed (baud rate or bits per second) of any serial interface used by the debugger for remote debugging.

-tty ***`device`***
: Run using *`device`* for your program's standard input and output.

2.2. Quitting the Debugger

quit

To exit the debugger, use the **quit** command (abbreviated **q**), or type an end-of-file character (usually **Ctrl**+**D**). If you do not supply *`expression`*, the debugger will terminate normally; otherwise, it will terminate using the result of *`expression`* as the error code.

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

2.3. Shell Commands

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

shell ***`command string`***

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

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

make ***`make-args`***

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

Chapter 3 *Debugger Commands*

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

3.1. Command Syntax

A debugger 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.

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

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

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

The debugger can also use **Enter** in another way: to partition lengthy output, in a way similar to the common utility **more** (see Section 14.4, “Screen Size” [120]). Since it is easy to press one **Enter** too many in this situation, the debugger disables command repetition after any command that generates this sort of display.

Any text from a **#** to the end of the line is a comment; it does nothing. This is useful mainly in command files (see Section 15.3, “Command Files” [126]).

3.2. Command Completion

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

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

```
(gdb) info bre Tab
```

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

```
(gdb) info breakpoints
```

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

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

```
(gdb) b make_ Tab
```

```
the debugger sounds bell; press Tab again, to see:
```

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

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

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

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

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

```
(gdb) b 'bubble(
Meta+?
bubble(double,double)    bubble(int,int)
(gdb) b 'bubble(
```

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

```
(gdb) b bub Tab
```

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

```
(gdb) b 'bubble(
```

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

3.3. Getting Help

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

help , **h**

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

```
(gdb) help
List of classes of commands:
running -- Running the program
stack -- Examining the stack
data -- Examining data
breakpoints -- Making program stop at certain points
files -- Specifying and examining files
status -- Status inquiries
support -- Support facilities
user-defined -- User-defined commands
aliases -- Aliases of other commands
obscure -- Obscure features
Type "help" followed by a class name for a list of
commands in that class.
Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

help *class*

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

```
(gdb) help status
Status inquiries.
List of commands:
show -- Generic command for showing things set
 with "set"
info -- Generic command for printing status
Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

help *command*

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

complete *args*

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

```
complete i
```

results in:

```
info
inspect
ignore
```

This is intended for use by Emacs.

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

info

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

set

You can assign the result of an expresson to an environment variable with **set**. For example, you can set the debugger prompt to a dollar sign with **set prompt $**.

show

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

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

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

show version

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

show copying

Display information about permission for copying the debugger.

show warranty

Display the GNU “NO WARRANTY” statement.

Chapter 4

Running Programs Under the Debugger

If you intend to run a program under the debugger, you must first generate debugging information when you compile it.

4.1. Compiling for Debugging

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

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

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

When you debug a program compiled with **-g -O**, remember that the optimizer is rearranging your code; the debugger shows you what is really there. Do not be too surprised when the execution path does not exactly match your source file! An extreme example:

if you define a variable, but never use it, the debugger never sees that variable—because the compiler optimizes it out of existence.

Some things do not work as well with **-g -O** as with **-g -O0**, particularly on machines with instruction scheduling. If in doubt, recompile with **-g -O0**, and if this fixes the problem, please report it to us as a bug (including a test case!).

4.2. Starting your Program

run ***`args`*** , **r** ***`args`***

> Use the **run** command to start your program under the debugger. You must first specify the program name with an argument to the debugger (see Chapter 2, *Getting In and Out of the Debugger* [7]), or by using the **file** or **exec-file** command (see Section 12.1, “Commands to Specify Files” [107]).

When you issue the **run** command, your program begins to execute immediately. See Chapter 5, *Stopping and Continuing* [23] for discussion of how to arrange for your program to stop. Once your program has stopped, you may call functions in your program, using the **print** or **call** commands. See Chapter 8, *Examining Data* [55].

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

Chapter 5 Stopping and Continuing

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

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

info program

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

5.1. Breakpoints, Watchpoints, and Exceptions

A *breakpoint* makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control in finer detail whether your program stops. You can set breakpoints with the **break** command and its variants (see Section 5.1.1, “Setting Breakpoints” [24]), to specify the place where your program should stop by line number, function name or exact address in the program.

A *watchpoint* is a special breakpoint that stops your program when the value of an expression changes. You must use a different command to set watchpoints (see Section 5.1.2, “Setting Watchpoints” [29]), but aside from that, you can manage a watchpoint like any other breakpoint: you enable, disable, and delete both breakpoints and watchpoints using the same commands.

You can arrange to have values from your program displayed automatically whenever the debugger stops at a breakpoint. See Section 8.6, “Automatic Display” [63].

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

5.1.1. Setting Breakpoints

Breakpoints are set with the **break** command (abbreviated **b**). The debugger convenience variable $bpnum records the number of the breakpoints you've set most recently; see Section 8.9, “Convenience Variables” [74], for a discussion of what you can do with convenience variables.

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

break *`function`*
: Set a breakpoint at entry to function *`function`*. When using source languages that permit overloading of symbols, such as C++, *`function`* may refer to more than one possible place to break. See Section 5.1.7, “Breakpoint Menus” [36], for a discussion of that situation.

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

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

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

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

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

break
: When called without any arguments, **break** sets a breakpoint at the next instruction to be executed in the selected stack frame (see Chapter 6, *Examining the Stack* [41]). In any selected frame but the innermost, this makes your program stop as soon as control returns to that frame. This is similar to the effect of a **finish** command in the frame inside the selected frame—except that **finish** does not leave an active breakpoint. If you use **break** without an argument in the innermost frame, the debugger stops the next time it reaches the current location; this may be useful inside loops.

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

break ... if *cond*
: Set a breakpoint with condition *cond*; evaluate the expression *cond* each time the breakpoint is reached, and stop only if the value is nonzero—that is, if *cond* evaluates as true. **...** stands for one of the possible arguments described above (or no argument) specifying where to break. See Section 5.1.5, “Break

Conditions” [32], for more information on breakpoint conditions.

tbreak *`args`*

Set a breakpoint enabled only for one stop. *`args`* are the same as for the **break** command, and the breakpoint is set in the same way, but the breakpoint is automatically deleted after the first time your program stops there. See Section 5.1.4, “Disabling Breakpoints” [31].

hbreak *`args`*

Set a hardware-assisted breakpoint. *`args`* are the same as for the **break** command and the breakpoint is set in the same way, but the breakpoint requires hardware support and some target hardware may not have this support. The main purpose of this is EPROM/ROM code debugging, so you can set a breakpoint at an instruction without changing the instruction. However the hardware breakpoint registers can only take two data breakpoints, and the debugger will reject this command if more than two are used. Delete or disable usused hardware breakpoints before setting new ones. See Section 5.1.5, “Break Conditions” [32].

thbreak *`args`*

Set a hardware-assisted breakpoint enabled only for one stop. *`args`* are the same as for the **hbreak** command and the breakpoint is set in the same way. However, like the **tbreak** command, the breakpoint is automatically deleted after the first time your program stops there. Also, like the **hbreak** command, the breakpoint requires hardware support and some target hardware may not have this support. See Section 5.1.4, “Disabling Breakpoints” [31]. Also See Section 5.1.5, “Break Conditions” [32].

rbreak *`regex`*

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

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

info breakpoints [*n*] , **info break [*n*]** , **info watchpoints [*n*]**

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

Breakpoint Numbers

Type
Breakpoint or watchpoint.

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

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

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

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

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

info break with a breakpoint number *n* as argument lists only that breakpoint. The convenience variable **$_** and the default examining-address for the **x** command are set to the address of the last breakpoint listed (see Section 8.5, “Examining Memory” [61]).

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

The debugger allows you to set any number of breakpoints at the same place in your program. There is nothing silly or meaningless about this. When the breakpoints are conditional, this is even useful (see Section 5.1.5, “Break Conditions” [32]).

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

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

maint info breakpoints

Using the same format as **info breakpoints**, display both the breakpoints you've set explicitly, and those the debugger is using for internal purposes. Internal breakpoints are shown with negative breakpoint numbers. The type column identifies what kind of breakpoint is shown:

breakpoint
Normal, explicitly set breakpoint.

watchpoint
Normal, explicitly set watchpoint.

longjmp
Internal breakpoint, used to handle correctly stepping through **longjmp** calls.

longjmp resume
Internal breakpoint at the target of a **longjmp**.

until
Temporary internal breakpoint used by the debugger **until** command.

finish
Temporary internal breakpoint used by the debugger **finish** command.

5.1.2. Setting Watchpoints

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

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

The debugger provides two special watchpoints that work at the full speed of the simulator. These are known as *hardware* breakpoints and will be used in preference to the much slower soft watchpoints.

watch *`expr`*

Set a watchpoint for an expression. The debugger will break when *`expr`* is written into by the program and its value changes. However, the hardware breakpoint registers can only take two data watchpoints, and both watchpoints must be the same kind. For example, you can set two watchpoints with **watch** commands, two with **rwatch** commands, *or* two with **awatch** commands, but you cannot set one watchpoint with one command and the other with a different command. the debugger will reject the command if you try to mix watchpoints. Delete or disable unused watchpoint commands before setting new ones.

rwatch *`expr`*

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

awatch *`expr`*

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

info watchpoints

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

5.1.3. Deleting Breakpoints

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

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

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

clear

Delete any breakpoints at the next instruction to be executed in the selected stack frame (see Section 6.3, "Selecting a Frame" [44]). When the innermost frame is selected, this is a good way to delete a breakpoint where your program just stopped.

clear *`function`* , **clear *`filename`*:*`function`***

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

clear *`linenum`* , **clear *`filename`*:*`linenum`***

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

delete [breakpoints] [*`bnums`*...]

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

5.1.4. Disabling Breakpoints

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

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

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

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

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

disable [breakpoints] [*bnums*...]

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

enable [breakpoints] [*bnums*...]

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

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

enable [breakpoints] delete *`bnums`*...
: Enable the specified breakpoints to work once, then die. The debugger deletes any of these breakpoints as soon as your program stops there.

Except for a breakpoint set with **tbreak** (see Section 5.1.1, “Setting Breakpoints” [24]), breakpoints that you set are initially enabled; subsequently, they become disabled or enabled only when you use one of the commands above. (The command **until** can set and delete a breakpoint of its own, but it does not change the state of your other breakpoints; see Section 5.2, “Continuing and Stepping” [37].)

5.1.5. Break Conditions

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

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

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

Break conditions can have side effects, and may even call functions in your program. This can be useful, for example, to activate functions that log program progress, or to use your own print functions to format special data structures. The effects are completely predictable unless there is another enabled breakpoint at the same address. (In that case, the debugger might see the other breakpoint first and stop your program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible for the purpose of performing side effects when a breakpoint is reached (see Section 5.1.6, “Breakpoint Command Lists” [34]).

Break conditions can be specified when a breakpoint is set, by using **if** in the arguments to the **break** command. See Section 5.1.1, “Setting Breakpoints” [24]. They can also be changed at any time with the **condition** command. The **watch** command does not recognize the **if** keyword; **condition** is the only way to impose a further condition on a watchpoint.

condition *`bnum`* *`expression`*

> Specify *`expression`* as the break condition for breakpoint or watchpoint number *`bnum`*. After you set a condition, breakpoint *`bnum`* stops your program only if the value of *`expression`* is true (nonzero, in C). When you use **condition**, the debugger checks *`expression`* immediately for syntactic correctness, and to determine whether symbols in it have referents in the context of your breakpoint. The debugger does not actually evaluate *`expression`* at the time the **condition** command is given, however. See Section 8.1, “Expressions” [56].

condition *`bnum`*

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

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

ignore *`bnum`* *`count`*

> Set the ignore count of breakpoint number *`bnum`* to *`count`*. The next *`count`* times the breakpoint is reached, your program's

execution does not stop; other than to decrement the ignore count, the debugger takes no action.

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

When you use **continue** to resume execution of your program from a breakpoint, you can specify an ignore count directly as an argument to **continue**, rather than using **ignore**. See Section 5.2, “Continuing and Stepping” [37].

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

You could achieve the effect of the ignore count with a condition such as **$foo-- <= 0** using a debugger convenience variable that is decremented each time. See Section 8.9, “Convenience Variables” [74].

5.1.6. Breakpoint Command Lists

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

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

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

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

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

Pressing **Enter** as a means of repeating the last the debugger command is disabled within a *`command-list`*.

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

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

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

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

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

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

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

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

```
cont
end
```

5.1.7. Breakpoint Menus

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

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

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

5.2. Continuing and Stepping

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

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

> Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument *`ignore-count`* allows you to specify a further number of times to ignore a breakpoint at this location; its effect is like that of **ignore** (see Section 5.1.5, “Break Conditions” [32]).
>
> The argument *`ignore-count`* is meaningful only when your program stopped due to a breakpoint. At other times, the argument to **continue** is ignored.
>
> The synonyms **c** and **fg** are provided purely for convenience, and have exactly the same behavior as **continue**.

To resume execution at a different place, you can use **return** (see Section 11.3, “Returning from a Function” [104]) to go back to the calling function; or **jump** (see Section 11.2, “Continuing at a Different Address” [103]) to go to an arbitrary location in your program.

A typical technique for using stepping is to set a breakpoint (see Breakpoints; watchpoints; and exceptions: Breakpoints.) at the beginning of the function or the section of your program where a problem is believed to lie, run your program until it stops at that breakpoint, and then step through the suspect area, examining the variables that are interesting, until you see the problem happen.

step

> Continue running your program until control reaches a different source line, then stop it and return control to the debugger. This command is abbreviated **s**.

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

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

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

step *`count`*

Continue running as in **step**, but do so *`count`* times. If a breakpoint is reached, stepping stops right away.

next [*`count`*]

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

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

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

finish

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

Contrast this with the **return** command (see Section 11.3, “Returning from a Function” [104]).

until, **u**

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

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

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

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

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

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

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

until ***`location`***, **u** ***`location`***
Continue running your program until either the specified location is reached, or the current stack frame returns. *`location`* is any of the forms of argument acceptable to **break** (see

Section 5.1.1, “Setting Breakpoints” [24]). This form of the command uses breakpoints, and hence is quicker than **until** without an argument.

stepi , **si**

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

It is often useful to do **display/i $pc** when stepping by machine instructions. This makes the debugger automatically display the next instruction to be executed, each time your program stops. See Section 8.6, “Automatic Display” [63].

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

nexti , **ni**

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

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

Chapter 6

Examining the Stack

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

Each time your program performs a function call, information about the call is generated. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a *stack frame*. The stack frames are allocated in a region of memory called the *call stack*.

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

One of the stack frames is *selected* by the debugger and many the debugger commands refer implicitly to the selected frame. In particular, whenever you ask the debugger for the value of a variable in your program, the value is found in the selected frame. There are special the debugger commands to select whichever frame you are interested in. See Section 6.3, “Selecting a Frame” [44].

When your program stops, the debugger automatically selects the currently executing frame and describes it briefly, similar to the

frame command (see Section 6.4, “Information about a Frame” [45]).

6.1. Stack Frames

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

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

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

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

frame *`args`*

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

select-frame

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

6.2. Backtraces

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

backtrace , **bt**

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

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

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

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

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

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

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

```
(gdb) bt 3
#0  whetstone.log10 (x=0.75) at whetstone.adb:190
#1  0xd24 in whetstone.log (x=0.75) at whetstone.adb:218
#2  0x124a in _ISTACK_SIZE () at whetstone.adb:404
```

```
#3  0x424 in main () at b~whetstone.adb:43
(gdb)
```

The display for frame zero does not begin with a program counter value, indicating that your program has stopped at the beginning of the code for line 226 of `whetstone.c`.

6.3. Selecting a Frame

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

frame *`n`* , f *`n`*

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

frame *`addr`* , f *`addr`*

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

up *`n`*

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

down *`n`*

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

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

For example:

```
(gdb) up
#1  0xd24 in whetstone.log (x=0.75) at whetstone.adb:218
218         return 2.302585093 * LOG10 ( X ) ;
(gdb)
```

After such a printout, the **list** command with no arguments prints ten lines centered on the point of execution in the frame. See Section 7.1, “Printing Source Lines” [47].

up-silently ***n*** , **down-silently** ***n***

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

6.4. Information about a Frame

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

frame , **f**

When used without any argument, this command does not change which frame is selected, but prints a brief description of the currently selected stack frame. It can be abbreviated **f**. With an argument, this command is used to select a stack frame. See Section 6.3, “Selecting a Frame” [44].

info frame , **info f**

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

- the address of the frame
- the address of the next frame down (called by this frame)
- the address of the next frame up (caller of this frame)

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

The verbose description is useful when something has gone wrong that has made the stack format fail to fit the usual conventions.

info frame *`addr`* , info f *`addr`*

Print a verbose description of the frame at address *`addr`*, without selecting that frame. The selected frame remains unchanged by this command. This requires the same kind of address (more than one for some architectures) that you specify in the **frame** command. See Section 6.3, “Selecting a Frame” [44].

info args

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

info locals

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

info catch

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

Chapter 7 *Examining Source Files*

The debugger can print parts of your program's source, since the debugging information recorded in the program tells the debugger what source files were used to build it. When your program stops, the debugger spontaneously prints the line where it stopped. Likewise, when you select a stack frame (see Section 6.3, “Selecting a Frame” [44]), the debugger prints the line where execution in that frame has stopped. You can print other portions of source files by explicit command.

If you use the debugger through its Emacs interface, you may prefer to use Emacs facilities to view source.

7.1. Printing Source Lines

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

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

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

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

list
: Print more lines. If the last lines printed were printed with a **list** command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (see Chapter 6, *Examining the Stack* [41]), this prints lines centered around that line.

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

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

set listsize ***`count`***

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

show listsize

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

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

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

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

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

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

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

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

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

list
: As described in the preceding table.

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

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

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

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

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

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

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

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

7.2. Searching Source Files

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

forward-search *`regexp`* , **search *`regexp`***

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

reverse-search *`regexp`*

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

7.3. Specifying Source Directories

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

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

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

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

directory *`dirname`* ...

dir *`dirname`* ...

Add directory *`dirname`* to the front of the source path. Several directory names may be given to this command, separated by **:** or whitespace. You may specify a directory that is already in the source path; this moves it forward, so the debugger searches it sooner.

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

directory

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

show directories

Print the source path: show which directories it contains.

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

1. Use **directory** with no argument to reset the source path to empty.

2. Use **directory** with suitable arguments to reinstall the directories you want in the source path. You can add all the directories in one command.

7.4. Source and Machine Code

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

info line *linespec*

Print the starting and ending addresses of the compiled code for source line *linespec*. You can specify source lines in any of the ways understood by the **list** command (see Section 7.1, “Printing Source Lines” [47]).

For example, we can use **info line** to discover the location of the object code for the first line of function `log10` in Whetstone:

```
(gdb) info line log10
Line 176 of "whetstone.adb" starts at address 0xc9a <whetstone__log10<
   and ends at 0xca4 <whetstone__log10+10>.
(gdb)
```

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

```
(gdb) info line *0xc9a
Line 176 of "whetstone.adb" starts at address 0xc9a <whetstone__log10<
   and ends at 0xca4 <whetstone__log10+10>.
(gdb)
```

After **info line**, the default address for the **x** command is changed to the starting address of the line, so that **x/i** is sufficient to begin examining the machine code (see Section 8.5, “Examining Memory” [61]). Also, this address is saved as the value of the convenience variable $_ (see Section 8.9, “Convenience Variables” [74]).

disassemble

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

We can use **disassemble** to inspect the object code range shown in the last **info line**.

```
(gdb) disassemble 0xc9a 0xca0
Dump of assembler code from 0xc9a to 0xca0:
0xc9a <whetstone__log10>:       sisp   r15,1
0xc9c <whetstone__log10+2>:     pshm   r14,r14
0xc9e <whetstone__log10+4>:     lr     r14,r15
End of assembler dump.
(gdb)
```

Chapter 8 *Examining Data*

The usual way to examine data in your program is with the **print** command (abbreviated **p**), or its synonym **inspect**. It evaluates and prints the value of an expression of the language your program is written in (see Chapter 9, *Using the Debugger with Different Languages* [79]).

print *`exp`* , print `/f` *`exp`*
: *`exp`* is an expression (in the source language). By default the value of *`exp`* is printed in a format appropriate to its data type; you can choose a different format by specifying **`/f`**, where *`f`* is a letter specifying the format; see Section 8.4, "Output Formats" [60].

print , print `/f`
: If you omit *`exp`*, the debugger displays the last value again (from the *value history*; see Section 8.8, "Value History" [72]). This allows you to conveniently inspect the same value in an alternative format.

A more low-level way of examining data is with the **x** command. It examines data in memory at a specified address and prints it in a specified format. See Section 8.5, "Examining Memory" [61].

If you are interested in information about types, or about how the fields of a struct or class are declared, use the **ptype *`exp`*** command rather than **print**. See Chapter 10, *Examining the Symbol Table* [95].

8.1. Expressions

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

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

Because C is so widespread, most of the expressions shown in examples in this manual are in C. See Chapter 9, *Using the Debugger with Different Languages* [79], for information on how to use expressions in other languages.

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

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

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

@
: **@** is a binary operator for treating parts of memory as arrays. See Section 8.3, “Artificial Arrays” [58], for more information.

::
: **::** allows you to specify a variable in terms of the file or function where it is defined. See Section 8.2, “Program Variables” [57].

{type} addr

Refers to an object of type *type* stored at address *addr* in memory. *addr* may be any expression whose value is an integer or pointer (but parentheses are required around binary operators, just as in a cast). This construct is allowed regardless of what kind of data is normally supposed to reside at *addr*.

8.2. Program Variables

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

Variables in expressions are understood in the selected stack frame (see Section 6.3, “Selecting a Frame” [44]); they must be either:

- global (or static)

or

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

This means that in the function

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

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

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

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

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

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

This use of **::** is very rarely in conflict with the very similar use of the same notation in C++. The debugger also supports use of the C++ scope resolution operator in the debugger expressions.

Warning Occasionally, a local variable may appear to have the wrong value at certain points in a function – just after entry to a new scope, and just before exit.

You may see this problem when you are stepping by machine instructions. This is because, on most machines, it takes more than one instruction to set up a stack frame (including local variable definitions); if you are stepping by machine instructions, variables may appear to have the wrong values until the stack frame is completely built. On exit, it usually also takes more than one machine instruction to destroy a stack frame; after you begin stepping through that group of instructions, local variable definitions may be gone.

8.3. Artificial Arrays

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

You can do this by referring to a contiguous span of memory as an *artificial array*, using the binary operator **@**. The left operand of **@** should be the first element of the desired array and be an individual object. The right operand should be the desired length of the array. The result is an array value whose elements are all of the type of the left argument. The first element is actually the left argument; the second element comes from bytes of memory immediately following those that hold the first element, and so on. Here is an example. If a program says

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

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

```
p *array@len
```

The left operand of **@** must reside in memory. Array values made with **@** in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions. Artificial arrays most often appear in expressions via the value history (see Section 8.8, “Value History” [72]), after printing one out.

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

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

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

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

Sometimes the artificial array mechanism is not quite enough; in moderately complex data structures, the elements of interest may not actually be adjacent—for example, if you are interested in the values of pointers in an array. One useful work-around in this situation is to use a convenience variable (see Section 8.9, “Convenience Variables” [74]) as a counter in an expression that prints the first interesting value, and then repeat that expression via **Enter**. For instance, suppose you have an array dtab of pointers to structures, and you are interested in the values of a field fv in each structure. Here is an example of what you might type:

```
set $i = 0
p dtab[$i++]->fv
Enter
Enter
...
```

8.4. Output Formats

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

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

x

Regard the bits of the value as an integer, and print the integer in hexadecimal.

d

Print as integer in signed decimal.

u

Print as integer in unsigned decimal.

o

Print as integer in octal.

t

Print as integer in binary. The letter **t** stands for “two”. [1]

[1] **b** cannot be used because these format letters are also used with the **x** command, where **b** stands for “byte”; see Section 8.5, “Examining Memory” [61].

a

Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located:

```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```

c
Regard as an integer and print it as a character constant.

f
Regard the bits of the value as a floating-point number and print using typical floating-point syntax.

For example, to print the program counter in hex (see Section 8.10, “Registers” [75]), type

```
p/x $pc
```

Note that no space is required before the slash; this is because command names in the debugger cannot contain a slash.

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

8.5. Examining Memory

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

x/*nfu* *addr* , **x *addr*** , **x**

Use the **x** command to examine memory.

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

`n`, the repeat count
: The repeat count is a decimal integer; the default is 1. It specifies how much memory (counting by units `u`) to display.

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

`u`, the unit size
: The unit size is any of

  b
  : Bytes.

  h
  : Halfwords (two bytes).

  w
  : Words (four bytes). This is the initial default.

  g
  : Giant words (eight bytes).

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

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

For example, **x/3uh 0x54320** is a request to display three halfwords (**h**) of memory, formatted as unsigned decimal integers (**u**), starting at address **0x54320**. **x/4xw $sp** prints the four words (**w**) of memory above the stack pointer (here, **$sp**; see Section 8.10, “Registers” [75]) in hexadecimal (**x**).

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

Even though the unit size *u* is ignored for the formats **s** and **i**, you might still want to use a count *n*; for example, **3i** specifies that you want to see three machine instructions, including any operands. The command **disassemble** gives an alternative way of inspecting machine instructions; see Section 7.4, “Source and Machine Code” [51].

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

The addresses and contents printed by the **x** command are not saved in the value history because there is often too much of them and they would get in the way. Instead, the debugger makes these values available for subsequent use in expressions as values of the convenience variables **$_** and **$__**. After an **x** command, the last address examined is available for use in expressions in the convenience variable **$_**. The contents of that address, as examined, are available in the convenience variable **$__**.

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

8.6. Automatic Display

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

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

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

display *`exp`*

Add the expression *`exp`* to the list of expressions to display each time your program stops. See Section 8.1, “Expressions” [56].

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

display/*`fmt`* *`exp`*

For *`fmt`* specifying only a display format and not a size or count, add the expression *`exp`* to the auto-display list but arrange to display it each time in the specified format *`fmt`*. See Section 8.4, “Output Formats” [60].

display/*`fmt`* *`addr`*

For *`fmt`* **i** or **s**, or including a unit-size or a number of units, add the expression *`addr`* as a memory address to be examined each time your program stops. Examining means in effect doing **x/*`fmt`* *`addr`***. See Section 8.5, “Examining Memory” [61].

For example, **display/i $pc** can be helpful, to see the machine instruction about to be executed each time execution stops (**$pc** is a common name for the program counter; see Section 8.10, “Registers” [75]).

undisplay *`dnums`*... , **delete display *`dnums`*...**

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

undisplay does not repeat if you press **Enter** after using it. (Otherwise you would just get the error **No display number ....**)

disable display *`dnums`*...

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

enable display *`dnums`*...

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

display

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

info display

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

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

8.7. Print Settings

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

These settings are useful for debugging programs in any language:

set print address , **set print address on**

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

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

set print address off

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

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

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

show print address

Show whether or not addresses are to be printed.

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

set print symbol-filename on

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

set print symbol-filename off

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

show print symbol-filename

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

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

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

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

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

show print max-symbolic-offset

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

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

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

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

Other settings control how different kinds of objects are printed:

set print array , **set print array on**

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

set print array off

Return to compressed format for arrays.

show print array

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

set print elements ***`number-of-elements`***

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

show print elements

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

set print null-stop

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

set print pretty on

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

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

set print pretty off
Cause the debugger to print structures in a compact format, like this:

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

This is the default format.

show print pretty

Show which format the debugger is using to print structures.

set print sevenbit-strings on

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

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

show print sevenbit-strings

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

set print union on

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

set print union off

Tell the debugger not to print unions which are contained in structures.

show print union

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

For example, given the declarations

```
typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly}
              Bug_forms;
struct thing {
  Species it;
  union {
    Tree_forms tree;
    Bug_forms bug;
  } form;
};
struct thing foo = {Tree, {Acorn}};
```

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

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

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

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

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

set print demangle , **set print demangle on**

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

show print demangle

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

set print asm-demangle , **set print asm-demangle on**

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

show print asm-demangle

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

set demangle-style *`style`*

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

auto
Allow the debugger to choose a decoding style by inspecting your program.

gnu
Decode based on the GNU C++ Compiler (**g++**) encoding algorithm. This is the default.

arm
Decode using the algorithm in the *C++ Annotated Reference Manual*.

foo
Show the list of formats.

show demangle-style

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

set print object , **set print object on**

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

set print object off
: Display only the declared type of objects, without reference to the virtual function table. This is the default setting.

show print object
: Show whether actual, or declared, object types are displayed.

set print static-members , **set print static-members on**
: Print static members when displaying a C++ object. The default is on.

set print static-members off
: Do not print static members when displaying a C++ object.

show print static-members
: Show whether C++ static members are printed, or not.

set print vtbl , **set print vtbl on**
: Pretty print C++ virtual function tables. The default is off.

set print vtbl off
: Do not pretty print C++ virtual function tables.

show print vtbl
: Show whether C++ virtual function tables are pretty printed, or not.

8.8. Value History

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

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

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

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

```
p *$
```

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

```
p *$.next
```

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

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

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

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

show values

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

show values *n*

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

show values +

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

Pressing **Enter** to repeat **show values *n*** has exactly the same effect as **show values +**.

8.9. Convenience Variables

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

Convenience variables are prefixed with **$**. Any name preceded by **$** can be used for a convenience variable, unless it is one of the predefined machine-specific register names (see Section 8.10, “Registers” [75]). (Value history references, in contrast, are *numbers* preceded by **$**. See Section 8.8, “Value History” [72].)

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

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

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

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

Convenience variables have no fixed types. You can assign a convenience variable any type of value, including structures and arrays, even if that variable already has a value of a different type. The convenience variable, when used as an expression, has the type of its current value.

show convenience

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

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

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

Repeat that command by typing **Enter**.

Some convenience variables are created automatically by the debugger 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, “Examining Memory” [61]). Other commands which provide a default address for x to examine also set $_ to that address; these commands include **info line** and **info breakpoint**. The type of $_ is void * except when set by the **x** command, in which case it is a pointer to the type of $__.

$__

The variable $__ is automatically set by the **x** command to the value found in the last address examined. Its type is chosen to match the format in which the data was printed.

$_exitcode

The variable $_exitcode is automatically set to the exit code when the program being debugged terminates.

8.10. Registers

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

info registers

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

info all-registers

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

info registers *`regname`* ...

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

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

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add four to the stack pointer[2] with

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

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

[2]This is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected; setting $sp is not allowed when other stack frames are selected. To pop entire frames off the stack, regardless of machine architecture, use the **return** command; see Section 11.3, “Returning from a Function” [104].

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

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

Normally, register values are relative to the selected stack frame (see Section 6.3, "Selecting a Frame" [44]). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the true contents of hardware registers, you must select the innermost frame (with **frame 0**).

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

Chapter 9

Using the Debugger with Different Languages

Language-specific information is built into the debugger for some languages, allowing you to express operations like the above in your program's native language, and allowing the debugger to output values in a manner consistent with the syntax of your program's native language. The language you use to build expressions is called the *working language*.

9.1. Switching Between Source Languages

There are two ways to control the working language—either have the debugger set it automatically, or select it manually yourself. You can use the **set language** command for either purpose. On startup, the debugger defaults to setting the language automatically. The working language is used to determine how expressions you type are interpreted, how values are printed, etc.

In addition to the working language, every source file that the debugger knows about has its own working language. For some object file formats, the compiler might indicate which language a particular source file is in. However, most of the time the debugger infers the language from the name of the file. The language of a source file controls whether C++ names are demangled—this way

backtrace can show each frame appropriately for its own language. There is no way to set the language of a source file from within the debugger.

This is most commonly a problem when you use a program, such as **cfront** or **f2c**, that generates C but is written in another language. In that case, make the program use **#line** directives in its C output; that way the debugger will know the correct language of the source code of the original program, and will display that source code, not the generated C code.

9.1.1. List of Filename Extensions and Languages

If a source file name ends in one of the following extensions, then the debugger infers that its language is the one indicated.

`.c`
: C source file

`.C`, `.cc`, `.cxx`, `.cpp`, `.cp`, `.c++`
: C++ source file

`.a`, `.ada`, `.adb`, `.ads`
: Ada source file

`.c66`, `.C66`, `.cor`, `.COR`
: Coral 66 source file

`.s`, `.S`
: Assembler source file. This actually behaves almost like C, but the debugger does not skip over function prologues when stepping.

9.1.2. Setting the Working Language

If you allow the debugger to set the language automatically, expressions are interpreted the same way in your debugging session and your program.

If you wish, you may set the language manually. To do this, issue the command **set language *`lang`***, where *`lang`* is the name of a language, such as **c**. For a list of the supported languages, type **set language**.

9.1.3. Having the Debugger Infer the Source Language

To have the debugger set the working language automatically, use **set language local** or **set language auto**. The debugger then infers the working language. That is, when your program stops in a frame (usually by encountering a breakpoint), the debugger sets the working language to the language recorded for the function in that frame. If the language for a frame is unknown (that is, if the function or block corresponding to the frame was defined in a source file that does not have a recognized extension), the current working language is not changed, and the debugger issues a warning.

This may not seem necessary for most programs, which are written entirely in one source language. However, program modules and libraries written in one source language can be used by a main program written in a different source language. Using the command **set language auto** in this case frees you from having to set the working language manually.

9.2. Displaying the Language

The following commands help you find out which language is the working language, and also which language source files were written in.

show language

Display the current working language. This is the language you can use with commands such as **print** to build and compute expressions that may involve variables in your program.

info frame

Display the source language for this frame. This language becomes the working language if you use an identifier from this frame. See Section 6.4, “Information about a Frame” [45], to identify the other information listed here.

info source

Display the source language of this source file. See Chapter 10, *Examining the Symbol Table* [95] to identify the other information listed here.

9.3. Supported Languages

The debugger supports Ada, C, C++, Coral 66 and assembly programming languages. Some the debugger features may be used in expressions regardless of the language you use: the debugger **@** and **::** operators, and the **{type}addr** construct (see Section 8.1, “Expressions” [56]) can be used with the constructs of any supported language.

The following sections detail to what degree each source language is supported by the debugger. These sections are not meant to be language tutorials or references, but serve only as a reference guide to what the debugger expression parser accepts, and what input and output formats should look like for different languages. There are many good books written on each of these languages; please look to these for a language reference or tutorial.

9.3.1. C and C++ operators

Operators must be defined on values of specific types. For instance, **+** is defined on numbers, but not on structures. Operators are often defined on groups of types.

For the purposes of C and C++, the following definitions hold:

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

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

,

The comma or sequencing operator. Expressions in a comma-separated list are evaluated from left to right, with the result of the entire expression being the last expression evaluated.

=

Assignment. The value of an assignment expression is the value assigned. Defined on scalar types.

***op*=**

Used in an expression of the form ***a op*= *b***, and translated to ***a* = *a op b***. ***op*=** and = have the same precendence. *op* is any one of the operators **|**, ^, **&**, <<, >>, +, -, *****, /, **%**.

?:

The ternary operator. ***a* ? *b* : *c*** can be thought of as: if *a* then *b* else *c*. *a* should be of an integral type.

||

Logical *or*. Defined on integral types.

&&

Logical *and*. Defined on integral types.

|

Bitwise *or*. Defined on integral types.

^

Bitwise exclusive-*or*. Defined on integral types.

&

Bitwise *and*. Defined on integral types.

==, !=

Equality and inequality. Defined on scalar types. The value of these expressions is 0 for false and non-zero for true.

<, >, <=, >=

Less than, greater than, less than or equal, greater than or equal. Defined on scalar types. The value of these expressions is 0 for false and non-zero for true.

<<, >>

left shift, and right shift. Defined on integral types.

@

The the debugger “artificial array” operator (see Section 8.1, “Expressions” [56]).

+, -
: Addition and subtraction. Defined on integral types, floating-point types and pointer types.

***, /, %**
: Multiplication, division, and modulus. Multiplication and division are defined on integral and floating-point types. Modulus is defined on integral types.

++, --
: Increment and decrement. When appearing before a variable, the operation is performed before the variable is used in an expression; when appearing after it, the variable's value is used before the operation takes place.

*****
: Pointer dereferencing. Defined on pointer types. Same precedence as **++**.

&
: Address operator. Defined on variables. Same precedence as **++**.

  For debugging C++, the debugger implements a use of **&** beyond what is allowed in the C++ language itself: you can use **&(&*ref*)** (or, if you prefer, simply **&&*ref***) to examine the address where a C++ reference variable (declared with **&*ref***) is stored.

-
: Negative. Defined on integral and floating-point types. Same precedence as **++**.

!
: Logical negation. Defined on integral types. Same precedence as **++**.

~
: Bitwise complement operator. Defined on integral types. Same precedence as **++**.

., ->
: Structure member, and pointer-to-structure member. For convenience, the debugger regards the two as equivalent, choosing whether to dereference a pointer based on the stored type information. Defined on **struct** and **union** data.

[]

Array indexing. ***a*[*i*]** is defined as ***(*a*+*i*)**. Same precedence as ->.

()

Function parameter list. Same precedence as ->.

::

C++ scope resolution operator. Defined on **struct**, **union**, and **class** types.

::

Doubled colons also represent the debugger scope operator (see Section 8.1, “Expressions” [56]). Same precedence as **::**, above.

9.3.2. C and C++ Constants

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

- Integer constants are a sequence of digits. Octal constants are specified by a leading **0** (i.e. zero), and hexadecimal constants by a leading **0x** or **0X**. Constants may also end with a letter **l**, specifying that the constant should be treated as a **long** value.
- Floating point constants are a sequence of digits, followed by a decimal point, followed by a sequence of digits, and optionally followed by an exponent. An exponent is of the form: **e[[+]|-]*nnn***, where *nnn* is another sequence of digits. The **+** is optional for positive exponents.
- Enumerated constants consist of enumerated identifiers, or their integral equivalents.
- Character constants are a single character surrounded by single quotes ('), or a number—the ordinal value of the corresponding character (usually its *ASCII* value). Within quotes, the single character may be represented by a letter or by *escape sequences*, which are of the form **\*nnn***, where *nnn* is the octal representation of the character's ordinal value; or of the form **\*x***, where ***x*** is a predefined special character—for example, **\n** for newline.
- String constants are a sequence of character constants surrounded by double quotes (").

- Pointer constants are an integral value. You can also write pointers to constants using the C operator **&**.
- Array constants are comma-separated lists surrounded by braces **{** and **}**; for example, **{1,2,3}** is a three-element array of integers, **{{1,2}, {3,4}, {5,6}}** is a three-by-two array, and **{&"hi", &"there", &"fred"}** is a three-element array of pointers.

9.4. The Ada Mode

The Ada mode of the debugger supports a fairly large subset of Ada expression syntax, with some extensions. The philosophy behind the design of this subset is:

- That the debugger should provide basic literals and access to operations for arithmetic, dereferencing, field selection, indexing, and subprogram calls.
- That type safety and strict adherence to Ada language restrictions are not particularly important to the the debugger user.
- That brevity is important to the the debugger user.

Thus, for brevity, the debugger acts as if there were implicit `with` and `use` clauses in effect for all user-written packages, making it unnecessary to fully qualify most names with their packages, regardless of context. Where this causes ambiguity, the debugger asks the user's intent.

The debugger will start in Ada mode if it detects an Ada main program. As for other languages, it will enter Ada mode when stopped in a program that was translated from an Ada source file.

While in Ada mode, you may use `--` for comments. This is useful mostly for documenting command files. The standard debugger comment (`#`) still works at the beginning of a line in Ada mode, but not in the middle (to allow based literals).

The debugger supports limited overloading. Given a subprogram call in which the function symbol has multiple definitions, it will use the number of actual parameters and some information about their types to attempt to narrow the set of definitions. It also makes very limited use of context, preferring procedures to functions in the context of the `call` command, and functions to procedures elsewhere.

9.4.1. Omissions from Ada

Here are the notable omissions from the subset:

- Only a subset of the attributes are supported:
- - 'First, 'Last, and 'Length on array objects (not on types and subtypes).
  - 'Min and 'Max.
  - 'Pos and 'Val.
  - 'Tag.
  - 'Range on array objects (not subtypes), but only as the right operand of the membership (`in`) operator.
  - 'Access, 'Unchecked_Access, and 'Unrestricted_Access (an extension).
  - 'Address.
- The names in `Characters.Latin_1` are not available and concatenation is not implemented. Thus, escape characters in strings are not currently available.
- The component-by-component array operations (and, or, xor, not, and relational and equality tests) are not implemented.
- There are no record or array aggregates.
- Dispatching subprogram calls are not implemented.
- The overloading algorithm is much more limited (that is less selective) than that of real Ada. It makes only limited use of the context in which a subexpression appears to resolve its meaning, and it is much looser in its rules for allowing type matches. As a result, some function calls will be ambiguous, and the user will be asked to choose the proper resolution.
- The operator new is not implemented.

- Entry calls are not implemented.
- Aside from printing, arithmetic operations on the native VAX floating-point formats are not supported.

9.4.2. Additions to Ada

As it does for other languages, the debugger makes certain generic extensions to Ada: the operators "`@`", "`::`", and {type} addr convenience variables and machine registers.

In addition, it provides a few other shortcuts and outright additions specific to Ada:

- The assignment statement is allowed as an expression, returning its right-hand operand as its value. Thus, you may enter

```
set x := y + 3
print A(tmp := y + 1)
```

- The semicolon is allowed as an "operator", returning as its value the value of its right-hand operand. This allows, for example, complex conditional breaks:

```
break f
condition 1 (report(i); k += 1; A(k) > 100)
```

- Rather than use catenation and symbolic character names to introduce special characters into strings, one may instead use a special bracket notation, which is also used to print strings. A sequence of characters of the form "`["XX"]`" within a string or character literal denotes the (single) character whose numeric encoding is XX in hexadecimal. The sequence of characters "`["""]`" also denotes a single quotation mark in strings. For example,

```
   "One line.["0a"]Next line.["0a"]"
```

  Contains an ASCII newline character (Ada.Characters.Latin_1.LF) after each period.

- The subtype used as a prefix for the attributes 'Pos, 'Min, and 'Max is optional (and is ignored in any case). For example, it is legal to write

```
print 'max(x, y)
```

- When printing arrays, the debugger uses positional notation when the array has a lower bound of 1, and uses a modified named notation otherwise. For example, a one-dimensional array of three integers with a lower bound of 3 might print as

```
(3 => 10, 17, 1)
```

  That is, in contrast to valid Ada, only the first component has a => clause.

- You may abbreviate attributes in expressions with any unique, multi-character subsequence of their names (an exact match gets preference). For example, you may use `a'len`, `a'gth`, or `a'lh` in place of `a'length`.

- Since Ada is case-insensitive, the debugger normally maps identifiers you type to lower case. The compiler uses upper-case characters for some of its internal identifiers, which are normally of no interest to users. For the rare occasions when you actually have to look at them, enclose them in angle brackets to avoid the lower-case mapping. For example,

```
(gdb) print <JMPBUF_SAVE>[0]
```

9.4.3. Stopping at the Beginning

The main procedure in Ada has no fixed name, and attempts to break on `main` will position you before elaboration. Therefore, Ada mode provides a convenient way to begin execution of the program and to stop at the beginning.

begin
: Does the equivalent of setting a temporary breakpoint at the beginning of the main procedure and then performing `run`.

Since in general there is package elaboration code that runs before the main procedure begins, it is possible that the program will stop before reaching the main procedure. However, the temporary breakpoint will remain to halt execution.

9.4.4. Breaking on Ada Exceptions

In Ada mode, you can set breakpoints that trip when your program raises selected exceptions.

info exceptions, **info exceptions regexp**

The `info exceptions` command permits the user to examine all defined exceptions within Ada programs. With a regular expression, regexp, as argument, prints out only those exceptions whose name matches regexp.

9.4.5. Extensions for Ada Tasks

Support for Ada tasks is analogous to that for threads. When in Ada mode (that is, when the "current language" is Ada), the debugger allows the following task-related commands:

info tasks

This command shows a list of current Ada tasks, as in the following example:

Example 9.1. Output from info tasks

```
(gdb) info tasks
 TCB      Task Task        Base Actv   On     Ready  Wakeup Time     Deadline
 Address    Id State       Prio Prio Hold     Count    (seconds)    (seconds)
+--------+----+----------+----+----+----+---------+------------+------------
*00010778    1 Running       0    0    0         3     0.000000     0.000000
 000107ba    2 At_Barrier    1    1    0         1     0.000000     0.000000
 00010ffa    3 Delayed      10   10    0        24     2.402170     0.000000
 0001183a    4 Delayed      10   10    0        24     2.402170     0.000000
+--------+----+----------+----+----+----+---------+------------+------------
```

In this listing, the asterisk before the first task indicates it to be the currently running task.

info queues

This command lists any tasks that are queued in the ready queue, the delay queue and the deadline queue.

Example 9.2. Output from info queues

```
(gdb) info queues
The ready queue is empty
Delay queue: 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
The deadline queue is empty
```

9.4.6. Debugging Generic Units

The compiler always uses code expansion for generic instantiation. This means that each time an instantiation occurs, a complete copy of the original code is made with appropriate substitutions.

It is not possible to refer to the original generic entities themselves in the debugger (there is no code to refer to), but it is certainly possible to debug a particular instance of a generic, simply by using the appropriate expanded names. For example, suppose that `Gen` is a generic package:

```
-- In file gen.ads:
generic package Gen is
   function F (v1 : in out INTEGER) return INTEGER;
end Gen;

-- In file gen.adb:
package body Gen is
   function F (v1 : in out INTEGER) return INTEGER is
   begin
      v1 := v1 + 1;
      return v1;         -- Line 5
   end F;
end Gen;
```

and we have the following expansions:

```
procedure G is
   package Gen1 is new Gen;
   package Gen2 is new Gen;
begin
   Gen1.F;
   Gen2.F;
   Gen1.F;
   Gen2.F;
end;
```

Then to break on a call to procedure `F` in the `Gen2` instance, simply use the command:

```
break G.Gen2.F
```

To break at a particular line in a particular generic instance, say the return statement in `G.Gen2`, append the line specification to the file and function name:

```
break gen.adb:G.Gen2.F:5
```

To break on this line line in *all* instances of `Gen`, use `` `*' `` as the function name:

```
break gen.adb:*:5
```

This will set individual breakpoints at all instances; they are independent of each other and you may remove, conditionalize, or otherwise modify them individually.

When a breakpoint occurs, you can step through the code of the generic instance in the normal manner. You can also examine values of data in the normal manner, providing the appropriate generic package qualification to refer to non-local entities.

9.4.7. Set commands for Ada

Ada introduces one new `set` command.

set varsize-limit size

Limit the size of the types of objects when those sizes are computed from run-time quantities to size bytes. When this is set to 0, there is no limit. By default, it is about 65K.

show varsize-limit

Show the limit on types whose size is determined by run-time quantities.

9.4.8. Known Peculiarities of Ada Mode

Besides the omissions listed previously (see Section 9.4.1, “Omissions from Ada” [87]), we know of several problems with and limitations of Ada mode in the debugger, some of which will be fixed with planned future releases of the debugger and the Ada compiler.

- Currently, the debugger has insufficient information to determine whether certain pointers represent pointers to objects or the objects themselves. Thus, the user may have to tack an extra .all after an expression to get it printed properly.
- Static constants that the compiler chooses not to materialize as objects in storage are invisible to the debugger.
- Renaming declarations are invisible to the debugger.
- Named parameter associations in function argument lists are ignored (the argument lists are treated as positional).
- Many useful library packages are currently invisible to the debugger.
- Fixed-point arithmetic, conversions, input, and output is carried out using floating-point arithmetic, and may give results that only approximate those on the host machine.
- The type of the 'Address attribute may not be System.Address.
- When stopped in a particular subprogram, you can access variables defined in other, lexically enclosing subprograms by their simple names. At the moment, however, this may not always work; it depends on whether the compiler happens to have made the necessary information (the "static link") available at execution time, which it can sometimes avoid. Of course, even in those cases where the compiler does not provide the information, you can still look at such variables by issuing the appropriate number of `up` commands to get to frame containing the variable you wish to see. Access to non-local variables does

not, at the moment, work in the test expressions for conditional breakpoints unless you happen to specify these while stopped in the subprogram in which they are to be applied.

Chapter 10

Examining the Symbol Table

The commands described in this section allow you to inquire about the symbols (names of variables, functions and types) defined in your program. This information is inherent in the text of your program and does not change as your program executes. The debugger finds it in your program's symbol table, in the file indicated when you started the debugger (see Section 2.1.1, “Choosing Files” [8]), or by one of the file-management commands (see Section 12.1, “Commands to Specify Files” [107]).

Occasionally, you may need to refer to symbols that contain unusual characters, which the debugger ordinarily treats as word delimiters. The most frequent case is in referring to static variables in other source files (see Section 8.2, “Program Variables” [57]). File names are recorded in object files as debugging symbols, but the debugger would ordinarily parse a typical file name, like `foo.c`, as the three words “`foo`” “`.`” “`c`”. To allow the debugger to recognize “`foo.c`” as a single symbol, enclose it in single quotes; for example,

```
p 'foo.c'::x
```

looks up the value of x in the scope of the file `foo.c`.

info address *`symbol`*

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

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

whatis *`exp`*

Print the data type of expression *`exp`*. *`exp`* is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place. See Section 8.1, “Expressions” [56].

whatis

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

ptype *`typename`*

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

ptype *`exp`* , ptype

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

For example, for this variable declaration:

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

the two commands give this output:

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

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

info types *regexp* , info types

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

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

info source

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

info sources

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

info functions

Print the names and data types of all defined functions.

info functions *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., excluding 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*.

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

set symbol-reloading on

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

set symbol-reloading off

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

show symbol-reloading

Show the current **on** or **off** setting.

maint print symbols *filename* , **maint print psymbols *filename*** , **maint print msymbols *filename***

Write a dump of debugging symbol data into the file *filename*. These commands are used to debug the the debugger symbol-reading code. Only symbols with debugging data are included. If you use **maint print symbols**, the debugger includes all the symbols for which it has already collected full details: that is, *filename* reflects symbols for only those files whose symbols the debugger has read. You can use the command **info sources** to find out which files these are. If you use **maint print psymbols** instead, the dump shows information about symbols that the debugger only knows partially — that is, symbols defined in files that the debugger has skimmed, but not yet read completely. Finally, **maint print msymbols** dumps just the minimal symbol information required for each object file from which the debugger has read some symbols. See Section 12.1, “Commands to Specify Files” [107], for a discussion of how the debugger reads symbols (in the description of **symbol-file**).

Chapter 11 Altering Execution

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

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

11.1. Assignment to Variables

To alter the value of a variable, evaluate an assignment expression. See Section 8.1, “Expressions” [56]. For example,

```
print x := 4
```

stores the value 4 into the variable x, and then prints the value of the assignment expression (which is 4). See Chapter 9, *Using the Debugger with Different Languages* [79], for more information on operators in supported languages.

If you are not interested in seeing the value of the assignment, use the **set** command instead of the **print** command. **set** is really the same as **print** except that the expression's value is not printed and is not put in the value history (see Section 8.8, “Value History” [72]). The expression is evaluated only for its effects.

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

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

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

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

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

To store values into arbitrary places in memory, use the **{...}** construct to generate a value of specified type at a specified address (see Section 8.1, “Expressions” [56]). For example, {int}0x83040 refers to memory location 0x83040 as an integer (which implies a certain size and representation in memory), and

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

stores the value 4 into that memory location.

11.2. Continuing at a Different Address

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

jump *`linespec`*

Resume execution at line *`linespec`*. Execution stops again immediately if there is a breakpoint there. See Section 7.1, "Printing Source Lines" [47], for a description of the different forms of *`linespec`*.

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

jump **`address`*

Resume execution at the instruction at address *`address`*.

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

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

makes the next **continue** command or stepping command execute at address 0x485, rather than at the address where your program stopped. See Section 5.2, "Continuing and Stepping" [37].

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

11.3. Returning from a Function

return , **return** ***`expression`***

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

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

This pops the selected stack frame (see Section 6.3, “Selecting a Frame” [44]), and any other frames inside of it, leaving its caller as the innermost remaining frame. That frame becomes selected. The specified value is stored in the registers used for returning values of functions.

The **return** command does not resume execution; it leaves the program stopped in the state that would exist if the function had just returned. In contrast, the **finish** command (see Section 5.2, “Continuing and Stepping” [37]) resumes execution until the selected stack frame returns naturally.

11.4. Calling Program Functions

call ***`expr`***

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

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

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

11.5. Patching Programs

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

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

set write on , **set write off**

> If you specify **set write on**, the debugger opens executable files for both reading and writing; if you specify **set write off** (the default), the debugger opens them read-only.
>
> If you have already loaded a file, you must load it again (using the **exec-file** command) after changing **set write**, for your new setting to take effect.

show write

> Display whether executable files are opened for writing as well as reading.

Chapter 12 The Debugger Files

The debugger needs to know the file name of the program to be debugged, both in order to read its symbol table and in order to start your program.

12.1. Commands to Specify Files

The usual way to specify an executable file name is with the command argument given when you start the debugger, (see Chapter 2, *Getting In and Out of the Debugger* [7]).

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

file `filename`

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

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

file

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

exec-file [*`filename`*]

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

symbol-file [*`filename`*]

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

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

The **symbol-file** command causes the debugger 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 the debugger.

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

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

The purpose of this two-stage reading strategy is to make the debugger start up faster. For the most part, it is invisible except for occasional pauses while the symbol table details for a particular source file are being read. (The **set verbose** command can turn these pauses into messages if desired. See Section 14.6, “Optional Warnings and Messages” [121].)

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

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

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

load *`filename`*

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

If you are using the simulator, then the program will be automatically loaded into the simualtor when you enter the **run** command. If you have selected the remote target, then you may use the **load** command to download the program. Note that the debugger may be used with a program that is already loaded.

load does not repeat if you press **Enter** again after using it.

section *`sect addr`*

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

info files , **info target**

> **info files** and **info target** are synonymous; both print the current target (see Chapter 13, *Specifying a Debugging Target* [113]), including the name of the executable file currently in use by the debugger, and the files from which symbols were loaded. The command **help target** lists all possible targets rather than current ones.

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

12.2. Errors Reading Symbol Files

While reading a symbol file, the debugger occasionally encounters problems, such as symbol types it does not recognize, or known bugs in compiler output. By default, the debugger does not notify you of such problems, since they are relatively common and primarily of interest to people debugging compilers. If you are interested in seeing information about ill-constructed symbol tables, you can either ask the debugger to print only one message about each such type of problem, no matter how many times the problem occurs; or you can ask the debugger to print more messages, to see how many times the problems occur, with the **set complaints** command (see Section 14.6, "Optional Warnings and Messages" [121]).

The messages currently printed, and their meanings, include:

`inner block not inside outer block in` *`symbol`*

> The symbol information shows where symbol scopes begin and end (such as at the start of a function or a block of statements). This error indicates that an inner scope block is not fully contained in its outer scope blocks.
>
> The debugger circumvents the problem by treating the inner block as if it had the same scope as the outer block. In the error message, *`symbol`* may be shown as "(don't know)" if the outer block is not a function.

`block at` *`address`* `out of order`

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

The debugger does not circumvent this problem, and has trouble locating symbols in the source file whose symbols it is reading. (You can often determine which source file is affected by specifying **set verbose on**. See Section 14.6, “Optional Warnings and Messages” [121])

`bad block start address patched`

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

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

`bad string table offset in symbol` *`n`*

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

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

`unknown symbol type 0x`*`nn`*

The symbol information contains new data types that the debugger does not yet know how to read. **0x*nn*** is the symbol type of the misunderstood information, in hexadecimal.

`stub type has NULL name`

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

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

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

`info mismatch between compiler and debugger`
: The debugger could not parse a type specification output by the compiler.

Chapter 13

Specifying a Debugging Target

A *target* is the execution environment occupied by your program. You can use the **target** command to specify one of the target types configured for the debugger (see Section 13.2, “Commands for Managing Targets” [113]).

13.1. Active Targets

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

Use the **exec-file** command to select a new executable target (see Section 12.1, “Commands to Specify Files” [107]).

13.2. Commands for Managing Targets

target ***`type parameters`***
: Connects the debugger host environment to a target machine.

The **target** command does not repeat if you press **Enter** again after executing the command.

help target

Displays the names of all targets available. To display targets currently selected, use either **info target** or **info files** (see Section 12.1, “Commands to Specify Files” [107]).

help target *`name`*

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

set gnutarget *`args`*

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

Note To specify a file format with **set gnutarget**, you must know the actual BFD name. See Section 12.1, “Commands to Specify Files” [107].

show gnutarget

Use the **show gnutarget** command to display what file format gnutarget is set to read. If you have not set gnutarget, the debugger will determine the file format for each file automatically and **show gnutarget** displays the following message:

```
The current BFD target is "auto".
```

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

target remote *`dev`*

Remote serial target in the debugger-specific protocol. The argument *`dev`* specifies what serial device to use for the connection (e.g. `/dev/ttya`). See Section 13.3, “Remote Debugging” [115]. **target remote** now supports the **load** command. This is only useful if you have some other way of getting the debug monitor to the target system, and you can put it somewhere in memory where it won't get clobbered by the download.

target sim

This is the target CPU simulator.

target remote

A target computer connected to the host computer by a serial interface.

13.3. Remote Debugging

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

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

Other remote targets may be available in your configuration of the debugger; use **help target** to list them.

Chapter 14 Controlling the Debugger

You can alter the way the debugger interacts with you by using the **set** command. For commands controlling how the debugger displays data, see Section 8.7, “Print Settings” [66]; other settings are described here.

14.1. Prompt

The debugger 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.

set prompt *`newprompt`*

Directs the debugger to use *`newprompt`* as its prompt string henceforth.

show prompt

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

14.2. Command Editing

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

You may control the behavior of command line editing in the debugger with the command **set**.

set editing , **set editing on**

Enable command line editing (enabled by default).

set editing off

Disable command line editing.

show editing

Show whether command line editing is enabled.

14.3. Command History

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

set history filename *`fname`*

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

set history save , **set history save on**

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

set history save off
Stop recording command history in a file.

set history size size

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

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

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

The commands to control history expansion are:

set history expansion on , **set history expansion**

Enable history expansion. History expansion is off by default.

set history expansion off
Disable history expansion.

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

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

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

show commands

Display the last ten commands in the command history.

show commands n
Print ten commands centered on command number n.

show commands +
Print ten commands just after the commands last printed.

14.4. Screen Size

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

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

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

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

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

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

14.5. Numbers

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

set input-radix *base*

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

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

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

set output-radix *base*

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

show input-radix

Display the current default base for numeric input.

show output-radix

Display the current default base for numeric display.

14.6. Optional Warnings and Messages

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

Currently, the messages controlled by **set verbose** are those that announce that the symbol table for a source file is being read; see **symbol-file** in Section 12.1, “Commands to Specify Files” [107].

set verbose on

Enables the debugger output of certain informational messages.

set verbose off
Disables the debugger output of certain informational messages.

show verbose

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

By default, if the debugger encounters bugs in the symbol table of an object file, it is silent; but if you are debugging a compiler, you may find this information useful (see Section 12.2, “Errors Reading Symbol Files” [110]).

set complaints limit

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

show complaints

Displays how many symbol complaints the debugger is permitted to produce.

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

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

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

set confirm off

Disables confirmation requests.

set confirm on
Enables confirmation requests (the default).

show confirm

Displays state of confirmation requests.

Chapter 15

Canned Sequences of Commands

Aside from breakpoint commands (see Section 5.1.6, “Breakpoint Command Lists” [34], the debugger provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files.

15.1. User-Defined Commands

A *user-defined command* is a sequence of the debugger commands to which you assign a new name as a command. This is done with the **define** command. User commands may accept up to 10 arguments separated by whitespace. Arguments are accessed within the user command via $arg0...$arg9. A trivial example:

```
define adder
  print $arg0 + $arg1 + $arg2
```

To execute the command use:

```
adder 1 2 3
```

This defines the command **adder**, which prints the sum of its three arguments. Note the arguments are text substitutions, so they may reference variables, use complex expressions, or even perform inferior functions calls.

define commandname

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

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

if

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

while

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

document *`commandname`*

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

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

help user-defined

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

show user , **show user** ***`commandname`***

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

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

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

15.2. User-Defined Command Hooks

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

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

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

```
define hook-stop
handle SIGALRM nopass
end
define hook-run
handle SIGALRM pass
end
define hook-continue
```

```
handle SIGLARM pass
end
```

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

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

15.3. Command Files

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

When you start the debugger, it automatically executes commands from its *init files*. These are files named `.gdbinit`. The debugger reads the init file (if any) in your home directory, then processes command line options and operands, and then reads the init file (if any) in the current working directory. This is so the init file in your home directory can set options (such as **set complaints**) which affect the processing of the command line options and operands. The init files are not executed if you use the **-nx** option; see Section 2.1.2, “Choosing Modes” [9].

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 debugger commands that normally print messages to say what they are doing omit the messages when called from command files.

15.4. Commands for Controlled Output

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

echo text

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

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

```
echo This is some text\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” [56], for more information on expressions.

output/fmt expression

Print the value of expression in format fmt. You can use the same formats as for **print**. See Section 8.4, “Output Formats” [60], 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 your program were to execute the C subroutine

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

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

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

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

Chapter 16 *Using History Interactively*

This chapter describes how to use the *History Library* interactively, from a user's standpoint.

16.1. History Interaction

The History Library provides a history expansion feature similar to the history expansion in csh. The following text describes the syntax you use to manipulate history information.

History expansion takes two parts. In the first part, determine which line from the previous history will be used for substitution. This line is called the *event*. In the second part, select portions of that line for inclusion into the current line. These portions are called *words*. The debugger breaks the line into words in the same way that the Bash shell does, so that several English (or UNIX) words surrounded by quotes are considered one word.

16.1.1. Event Designators

An *event designator* is a reference to a command line entry in the history list.

!
: Start a history substitution, except when followed by a space, tab, or the end of the line... **=** or **(**.

!!
: Refer to the previous command. This is a synonym for **!-1**.

!n
: Refer to command line n.

!-n
: Refer to the command line n lines back.

!string
: Refer to the most recent command starting with string.

!?string[**?**]
: Refer to the most recent command containing string.

16.1.2. Word Designators

A **:** separates the event designator from the *word designator*. It can be omitted if the word designator begins with a ^, **$**, ***** or **%**. Words are numbered from the beginning of the line, with the first word being denoted by a 0 (zero).

0 (zero)
: The zero'th word. For many applications, this is the command word.

n
: The n'th word.

^
: The first argument. that is, word 1.

$
: The last argument.

%
: The word matched by the most recent **?string?** search.

x-y
: A range of words; **-y** Abbreviates **0-y**.

*****

All of the words, excepting the zero'th. This is a synonym for **1-$**. It is not an error to use ***** if there is just one word in the event. The empty string is returned in that case.

16.1.3. Modifiers

After the optional word designator, you can add a sequence of one or more of the following *modifiers*, each preceded by a **:**.

#

The entire command line typed so far. This means the current command, not the previous command.

h

Remove a trailing pathname component, leaving only the head.

r

Remove a trailing suffix of the form **.**suffix, leaving the basename.

e

Remove all but the suffix.

t

Remove all leading pathname components, leaving the tail.

p

Print the new command but do not execute it.

Index