A Guide to the gdb Debugger

Timothy A. Gonsalves
TeNeT Group
Dept of Computer Science & Engineering
Indian Institute of Technology, Madras - 600 036
E-mail: tag@ooty.TeNeT.res.in

July 1999

Contents

1 Introduction

A common approach to debugging a program is to insert print statements at various locations. This has the disadvantages that the code becomes harder to read as more such statements are inserted, and that the debugging output can become unmanageably voluminous. After the program is debugged, while removing some of these print statements, the program may get accidentally changed.

A *symbolic debugger* such as *gdb* allows you to concentrate your attention on the small fraction of the program that typically has most of the bugs. The debugger allows you to control execution of the program, suspending the execution at certain lines. You can then examine and even change the values of any variables in the program. All this is done without touching the source code.

Indeed, the most powerful method of debugging is single-stepping line-by-line through the code, examining the values of key variables after every statement. It is only with a symbolic debugger that this is possible.

1.1 Typographical Conventions

We use the following conventions in this guide:

`emacs`	the name of a specific command or file
file	you should replace *file* with a specific name
gdb`> help quit`	at the *gdb* prompt, type the command `help quit`, then press the <ENTER> key
Exit gdb.	output that you see on the screen

2 Getting Started

First, compile your program using the `-g` option. This instructs the compiler to put information about the symbols – functions and variables – into the executeable file for use by the debugger. Next, invoke *gdb* with the name of the program:

```
% gcc -g -o myprog myprog.c
% gdb myprog
gdb>
```

To exit from *gdb*, type `quit`. For a list of commands, type `help`.

In the sections below, we describe some of the more useful *gdb* commands. Although all commands are full words, you can type any unambiguous prefix. For instance, `cont` instead of `continue`. You can also define short *aliases* for frequently-used commands (see section 7).

2.1 Getting Help

Use the `help` command in *gdb* for a list of topics. Then type `help` ***`topic`*** for details on a specific topic. `help` *`cmd`* gives help on the command `cmd`. For comprehensive coverage of *gdb*, read the Info pages. At the shell prompt, type `info`. When `info` starts, type `mgdb`<ENTER>). Alternatively, if you use *Emacs*, type the Emacs command `<Ctrl-h i>` to start *Info*.

3 Program Control

break *nnn*	set a break-point – program execution stops when it reaches line *nnn*
break *func*	set a break-point – program execution stops when it enters function *func*
run	start program execution.
continue	continue execution from a break-point
step	execute the next source line and then break. Steps into function calls.
next	execute the next source line and then break. Steps over function calls.
finish	execute until the end of the current function
<ENTER>	pressing <ENTER> by itself repeats the last command
watch *expr*	execution breaks whenever the expression *expr* changes value

Before running the program, set some break-points. The `run` command optionally takes command-line arguments and I/O redirection:

```
gdb> run -n *.c < infile
```

is equivalent to typing `myprog -n *.c < infile` at the shell prompt.

`next` treats a function call as a single line, while `step` allows you to debug inside the function also. If you inadvertently step into a function, use `finish` to execute to its return without breaking. To single-step throug several lines, type `next` the repeatedly type <ENTER>.

A watchpoint is useful to catch invalid data values. Example, if `n` should never exceed 100, the set a watchpoint: `watch n > 100`.

4 Data Access

When *gdb* stops execution at a break-point or after a `next` or `step`, you can examine and modify variables in the program.

print x	displays the value of the variable `x`
print *node	displays the fields of the structure `node`
print F(a+b)	calls the function `F()` with argument `a+b` and prints its return value
set x=10	sets the variable `x` to 10

You can print the value of any arbitrary expression in the language of the program being debugged. By calling a function with a variety of arguments, you can test the function without having to write testing code.

5 Source Files

list *nnn*	lists a window of 10 lines around line *nnn* in the source file
search *pat*	searches forward for the next source line containing the regular-expression *pat*
reverse-search	like `search`, but searches backwards

The search commands without an argument use the previous argument. Note that a simple form of regular expression is a string. Example, `search MyFunc` will search for the next call to MyFunc or its definition.

6 Core Dumps

If your program, `myprog`, suffers a serious exception such as a segmentation fault (usually a mis-behaving pointer in C/C++) and produces a `core` dump, you can use *gdb* to examine the state of the program at the time of the core dump. Start *gdb* with:

```
% gdb myprog core
```

When *gdb* starts, it tells you the line number in which the exception occurred. You can use `print` to examine any variables. To see how the program got to that line, use the command `bt` (short for `backtrace`). This shows all function calls from `main()` to the function in which the fault occurred.

7 Controlling gdb

define *cmd*	define *cmd* your own commands
show user	list user-defined commands
show	displays information about *gdb*. `help show` for details

info displays information about the program being debugged. `help info` for details

.gdbinit a *gdb* command file

Use `define` to make short versions of commonly-used commands. For example:

```
gdb> define p1
Type commands for definition of ”p1”.
End with a line saying just ”end”.
> print aLongVariableName->field.data
> end
gdb> p1
2354
gdb>
```

When *gdb* starts, it automatically reads `/.gdbinit` if it exists, then `.gdbinit` in the current directory, if it exists. Thus, you can customise *gdb* to your liking, and also customise it for each project (assumed to be in a separate sub-directory). These command files can contain any *gdb* commands as you would type them in *gdb*. Any line starting with a `#` is a comment, and blank lines are ignored.

8 Advanced Topics

This guide merely scratches the surface of what is possible with *gdb*. Most of the commands described here have many more features. *gdb* also has many more features such as allowing you to debug an already running process, to handle signals, to step at the machine instruction level. Use the `help` command in *gdb* and read the Info pages (see section 2.1).