Debugging with `gdb`

Luis Fernandes
Department of Electrical and Computer Engineering
Ryerson Polytechnic University

March 19, 2001

To err is human...
– Alexander Pope, 1688-1744

1 Introduction

`gdb`, the GNU debugger, is a source-level debugger for programs written in C, C++ and Modula-2. The program being debugged is run under the control of the debugger permitting its execution to be halted at any point, permitting the execution of a single line of code and permitting the contents of variables to be displayed and modified before execution continues, thus catching bugs as they happen.

Source-code files to be debugged via `gdb` must be compiled with `-g` option[1].

`gdb` may be run outside of Emacs, however, it is highly recommended that the edit-compile-debug cycle be performed within Emacs.

2 Debugging with `gdb` within Emacs

2.1 Invoking `gdb`

To invoke the debugger from within Emacs, type: `M-x gdb RET`, you will then be prompted for the name of the binary executable to be debugged; e.g. to debug the program `towers`, type: `M-x gdb RET towers RET` (`towers` is the name of the executable program).

[1]Use `CFLAGS = -DDEBUG -g` in your `Makefile`

2.2 `break`: Setting breakpoints

Upon entering the debugger, set one or more breakpoints letting the debugger know where to stop execution. To set a breakpoint at the function `main()`, type: `break main` (`main()` is a good place to set a breakpoint; any other function name may be substituted if you have localized the bug; e.g. `break towers`).

2.3 `run`: Running the program

To begin execution of the the program via the debugger, type: `run`. Command-line parameters (if any) are passed as arguments to the `run` command; e.g. `run 3 1 2` is equivalent to typing `towers 3 1 2` in an `xterm`. If the program reads a file from `stdin`, use `run < file`.

At this point, Emacs will display the source-code in another window, and will indicate the next line to be executed by displaying a arrow (`=>`).

2.4 `step` and `next`: Stepping the program

To single-step a single line of code, type: `step` (abbreviated `s`). The line of code will execute and the `=>` arrow will move to indicate the next line to be executed; the `step` command will step *into* functions.

A single line of code may also be executed by typing: `next` (abbreviated `n`); the `next` command will step *over* functions (i.e. the `gdb` will execute the function to completion and stop at the line immediately following the function call). Note that `s 5` (or `n 5`) may be used to step 5 lines.

2.5 `continue` and `finish`: Controlling execution

To continue executing until the next break-point, type: `continue` (abbreviated `c`).

Within a function, typing: `finish` (abbreviated `fin`) will execute the rest of the function and stop at the next line immediately following the function call.

2.6 `set` and `print`: Setting & examining variables

To print the contents of a variable, type: `print` (abbreviated `p` followed by the variable name; e.g. `p argc`. If `s` is a character pointer, `p s` will print the address of the pointer the contents of the variable. The contents of data structures may also be examined; e.g. `p state[2].name`.

The value of a variable may be changed using the `set` command; e.g. `set i=5` thus allowing you to expriment with the conditions necessary to fix a particular bug.

2.7 Miscellaneous commands

The previous `gdb` command may be re-executed by typing Return.

`M-p` and `M-n` provide command history.

The `help` command provides command-usage information for all the `gdb` commands; e.g. `help break`.

To abort the current debugging session, but remain within the debugger, type: `kill`. Typically this is done just before re-compiling the program and re-running it.

To re-compile the program from within `gdb`, type: `make` (provided there is a Makefile).

To re-start debugging from the beginning, type: `run`. Command-line arguments passed to the initial `run` command will be re-used.

To exit the debugger, type: `quit`.

3 Further reading

The commands introduced here are sufficient to debug most programs. To learn about additional `gdb` commands read the `gdb` manual page. The Emacs built-in online help browser also has information on using gdb (`M-x info`).

Debugging with GDB: The GNU Source-Level Debugger, Richard M. Stallman. Free Software Foundation, 1998, (`http://www.gnu.org/doc/doc.html`).

If you prefer a GUI debugger, try `ddd`, a frontend to `gdb`.

4 Command Summary

gdb Command	Abbrev.	Action
`break` *func* or #	`b` *func* or #	set breakpoint in *func* or at line #
`print` *var*	`p` *var*	print contents of *var*
`run` *args*	`r` *args*	run program with (optional) *args*
`step` #	`s` #	execute into (optional #) line(s)
`next` #	`n` #	execute over (optional #) line(s)
`continue`	`c`	continue execution until next breakpoint
`finish`	`fin`	finish execution of current function
`set` *var*=*val*	`set` *var*=*val*	set contents of *var* to *val*
`help` *cmd*	`h` *cmd*	get additional info on command *cmd*
`quit`	`q`	quit the debugger *cmd*

Table 1: `gdb` *Commands.*