

A Tutorial on the Intermediate Use of GDB

Travis Spencer

CS Tutors – Portland State University

July 26, 2005

1 Introduction

A debugger is one of the most valuable tools at a programmer's disposal. A good debugger makes it possible to find subtle bugs, sidesteps the need to litter code with a barrage of print statements, and alleviates the need for guesswork associated with coding complex applications. One such debugger that is extremely powerful is GDB, the GNU project's source code debugger. While it is a simple tool that can be used almost immediately, mastering its advanced features takes time and experience. This tutorial will expose you to some of these features.

2 Some Advanced Commands

For starters, GDB provides many commands that aren't necessary when debugging simple programs; however, as the complexity of the problems increases, you may find yourself facing difficult situations for which the debugger doesn't provide as much help as you need. For instance, at some point you've probably had to debug a piece of code that contains a loop that will execute another hundred or so time before it terminates, and you are no longer interested in the loop's body? Rather than setting another breakpoint just beyond the end of the loop or worse — actually stepping through the loop till it terminates — you can execute the `until` command which will cause the debugger to continue *until* the current loop completes. No need for a temporary breakpoint; no need to find out the line number of the state-

ment just beyond the loop's body; no need to step through hundreds of lines of code unnecessarily!

The `until` command and some other more advanced commands are listed below with their abbreviated form, expected arguments, and a description:

rbreak *regex* (sb) Sets a breakpoint on all functions matching a given regular expression.

tbreak *args* (tb) Sets a temporary breakpoint that will cause the program to halt a single time. The `args` parameter can be any value that can be given to the `break` command.

until (u) This command can be used to avoid stepping through a loop repeatedly. If it is executed in a loop, GDB will continue until the loop completes.

finish (fin) This command will instruct GDB to step out of the current frame and execute it (as opposed to a `return` that will immediately exit the frame without running the code).

condition breakpoint expression (con) Adds or deletes a condition of a particular breakpoint. If a logical expression is provided, it is associated with the specified breakpoint number (which is obtained by issuing an `info break` command). If no expression is given, the associated break condition is removed.

search *regex* (fo) Searches forward through the source code for a regular expression.

reverse-search *regex* (rev) Searches backward for a given regular expression.

show values Output the last 10 values that were displayed with a `print` statement.

3 Variables

GDB allows you to create variables that you can later refer to. Variables begin with a '\$' symbol and cannot conflict with any predefined variable names (e.g., register names). This allows you to save output from `print`

statement for convenience, create counters, or whatever. For example, to loop through an array of structures and see what the value of some field is, you might execute these GDB commands:

```
gdb> set $i = 0
gdb> p foo[$i++]>fieldValue
```

By simply hitting return repeatedly, `$i` will be incremented, and the field's value of successive structures in the array will be printed.

4 Command History

Like many modern shells (including `tcsh` and `bash`), GDB has a history feature that will allow you to quickly repeat previously entered commands or parts of them. Rather than retyping the arguments you gave to the last command, for example, you can simply use the `!*` event designator. There are many such designators; for a list see `Event Designators` in the `bash(1)` man page.

Before you can use GDB's history capabilities, you must enable it. It is off by default because some of the event designators can be confused with the logical operators `!` and `!=`. For example, does `p !*-100` mean print the difference of the last command's arguments and one hundred or the negation of the value at the address negative one hundred? This parsing dilemma can be sidestepped if you always follow the logical operators with a space.

If you think the benefits of using command history outweigh the potential for parsing problems, you need to enable it with the command `set history expansion on`. Rather than doing this every time you start GDB, you can add it to the file `.gdbinit` in your home directory.

5 Setting Variable Values

When you are running GDB, you can change the value of any variable in the scope of the current frame. This allows you to test values without having to recompile. For instance, if the value of some variable is causing problems that you suspect is the root of your trouble, you can simply change it by issuing the command `print var=newvalue` where `var` is the variable that you want to change and `newvalue` is whatever value you want `var` to have. This

technique is especially helpful when returned the application to its current state and location would be difficult or time consuming after a restart.

6 Changing argv on the Fly

To pass arguments to a program that is being debugged, normally you tack them on the end of a run command. For example, to pass your program the value “foo”, you would issue the command `run foo`. However, if you start your program by simply typing `run` or `r`, and you later want to debug the application as if an argument had been passed to it, you don’t have to restart the application. Rather, you can use the `set args` command. This command will set `argv` and `argc` as if they had been set initially.

7 Expanding Macros

Have you ever needed to know what the value of a macro is while debugging, but it wasn’t available because it had been removed during preprocessing. It is possible actually. To find out the value of a macro, you need to first recompile your code with debugging set to high (i.e., `-g3`) and you need to include the `-gdwarf-2` flag. So you will have a compilation command such as `gcc -gdwarf-2 -g3 sample.c -o sample`. Now, while debugging you can print the value of a macro such as `MY_MACRO` by entering the command `info macro MY_MACRO`.

8 Debugging Multiple Processes

On the Solaris machines, GDB doesn’t provide any special support for debugging programs with multiple processes; however, on the Linux machines in the lab (and any Linux box running a kernel newer than 2.5.60), GDB does have the capability to step into the child process after forking and execing. To step into a child process, the debugger must first be put into follow fork child mode. This is done by executing a statement such as this in GDB: `follow-fork-mode child`. After putting the debugger in this mode (which can be verified by executing the command `show follow-fork-mode`), break points in the child process’s code will cause GDB the halt just as in the parent.

A catchpoint can be used to make GDB halt whenever a `fork`, `vfork`, or `exec` is called as long as the debugger is in follow fork child mode. For instance, to cause execution to stop whenever an `exec` is called, set a catchpoint like this: `catch exec`. Afterward, every time a child process is spawned, GDB will stop. Likewise, to cause the debugger to stop whenever a `fork` or `vfork` is called, execute `catch fork` or `catch vfork` respectively. For more information on catchpoints, see the `gdb` documentation.

9 Using GDB with Emacs

Using GDB from the command line can be a little terse. If the plainness and compactness of GDB doesn't bode well with you, you may find the interface to GDB provided through Emacs a little more natural and usable. In Emacs, you start the debugger by executing the commands `M-x gdb`. In the minibuffer, Emacs prompts you for the name of the executable. After typing it in, hit return.¹ A new buffer will be opened containing the normal GDB startup info and a prompt where you can interact with the debugger like usual. After setting a breakpoint and putting GDB into run mode, however, the differences become very obvious.

When GDB hits the breakpoint, it opens a second window containing the source code at that particular place like this for example:

```
int main(int argc, char** argv)
{
B   int testval = argv && argv[1] ? atoi(argv[1]) : DEFAULT_VALUE;
=> int result = setjmp(buf);

    if (result)
        printf("fib of %d = %d\n", testval, result);
    else
        fib(testval);
}
```

The 'B' indicates a breakpoint and is only present if you are using Emacs 22 or newer; the '=' indicates the line that is about to be executed. It will advance every time you instruct GDB to step forward using `step`, `next`,

¹Starting in version 22, Emacs will guess the name of the executable for you.

`continue`, etc; however, it will never be saved when the buffer is written out to disk.

The buffer containing the source code is like any other buffer. It can be edited, searched, and compared with the current version in a source code control system's repository all while the debugger is running. If changes are made to the code, recompile it, jump to the GDB buffer, and restart the debugger. Then, the changes will take effect. For more information on using GDB with emacs, see the debuggers section of the emacs info manual (i.e., run `info emacs debuggers`).

10 Conclusion

As the complexity of programs increases, knowledge of GDB's advanced features must also increase. With an understanding of the more complex capabilities of the GNU debugger, finding hidden bugs becomes a lot less tedious and time consuming. With the aid of new commands, variables, history and emacs, big problems become a lot more manageable. This tutorial is brief, and it hasn't even touched on the extent of most of GDB's advanced features. For more information, check the `gdb` on-line documentation.