

Princeton University

COS 217: Introduction to Programming Systems

GDB Tutorial and Reference

Part 1: Tutorial

This tutorial describes how to use a minimal subset of the `gdb` debugger. For more information see Part 2 of this document and the online `gdb` tutorial at <http://sourceware.org/gdb/current/onlinedocs/gdb/>.

The tutorial assumes that you've created files named `testintmath.c`, `intmath.h`, and `intmath.c` in your working directory, containing the (version 4) program recently discussed in precepts. Those files are available through the course *Schedule* Web page.

Introduction

Suppose you're developing the `testintmath` (version 4) program. Further suppose that the program preprocesses, compiles, assembles, and links cleanly, but produces incorrect results at run-time. What can you do to debug the program?

One approach is temporarily to insert calls to `printf(...)` or `fprintf(stderr, ...)` throughout the program to get a sense of the flow of control and the values of variables at critical points. That's fine, but often is inconvenient.

An alternative is to use `gdb`. `gdb` is a powerful debugger. It allows you to set breakpoints in your program, step through your executing program one line at a time, examine the values of variables at breakpoints, examine the function call stack, etc.

Building

To prepare to use `gdb`, build your program with the `-g` option:

```
$ gcc217 -g testintmath.c intmath.c -o testintmath
```

The `-g` option tells `gcc217` to place extra information in the `testintmath` file that `gdb` uses.

It's a common error to forget to specify the `-g` option when building in preparation for using `gdb`.

Running gdb

The next step is to run `gdb`. You can run `gdb` directly from the shell, but it's much better to run it from within `emacs`. So launch `emacs`, with no command-line arguments:

```
$ emacs
```

Now call the `emacs` `gdb` function via these keystrokes:

```
<Esc key> x gdb <Enter key>
```

The `emacs` editor displays the message:

```
Run gud-gdb (like this): gdb --fullname
```

followed by the name of some executable binary file. If that name is not `testintmath`, then use the backspace key to delete it, and type `testintmath`. Then type the Enter key.

At this point you're executing `gdb` from within `emacs`. `gdb` is displaying its `(gdb)` prompt.

Running Your Program

Issue the `run` command to run the program:

```
(gdb) run
```

Enter 8 as the first integer, and 12 as the second integer. `gdb` runs the program to completion, indicating that the Program exited normally.

File redirection is specified as part of the `run` command. For example, the command `run < somefile` runs the program, redirecting its standard input to `somefile`.

Command-line arguments are specified as part of the `run` command. For example, the command `run arg1 arg2` runs the program with command-line arguments `arg1` and `arg2`. The `testintmath` program ignores its command-line arguments; of course other programs do not.

Using Breakpoints

Set a breakpoint at the beginnings of some functions using the `break` command:

```
(gdb) break main
(gdb) break IntMath_gcd
```

Another way to set a breakpoint is by specifying a file name and line number separated by a colon, for example, `break intmath.c:20`.

Then run the program:

```
(gdb) run
```

`gdb` pauses execution near the beginning of `main()`. It opens a second window in which it displays your source code, with an arrow pointing to the about-to-be-executed line.

Issue the `continue` command to tell command `gdb` to continue execution past the breakpoint:

```
(gdb) continue
```

`gdb` continues past the breakpoint at the beginning of `main()`, and execution is paused at a call of `scanf()`. Enter 8 as the first number. Execution is paused at the second call of `scanf()`. Enter 12 as the second number. `gdb` is paused at the beginning of `IntMath_gcd()`.

Then issue another `continue` command:

```
(gdb) continue
```

Note that `gdb` is paused, again, at the beginning of `IntMath_gcd()`. (Recall the `IntMath_gcd()` is called twice: once by `main()`, and once by `IntMath_lcm()`.)

While paused at a breakpoint, issue the `kill` command to stop execution:

```
(gdb) kill
```

Type `y` to confirm that you want `gdb` to stop execution.

Issue the `clear` command to get rid of a breakpoint:

```
(gdb) clear IntMath_gcd
```

At this point only one breakpoint remains: the one at the beginning of `main()`.

Stepping Through the Program

Run the program again:

```
(gdb) run
```

Execution pauses at the beginning of `main()`. Issue the `next` command to execute the next line of your program:

```
(gdb) next
```

Continue issuing the `next` command repeatedly until the program ends.

Run the program again:

```
(gdb) run
```

Execution pauses at the beginning of `main()`. Issue the `step` command to execute the next line of your program:

```
(gdb) step
```

Continue issuing the `step` command repeatedly until the program ends. Is the difference between `next` and `step` clear? The `next` command tells `gdb` to execute the next line, while staying at the same function call level. In contrast, the `step` command tells `gdb` to step into a called function.

Examining Variables

Set a breakpoint at the beginning of `IntMath_gcd()`:

```
(gdb) break IntMath_gcd
```

Run the program until execution reaches that breakpoint:

```
(gdb) run  
(gdb) continue
```

Now issue the `print` command to examine the values of the parameters of `IntMath_gcd()`:

```
(gdb) print iFirst  
(gdb) print iSecond
```

In general, when paused at a breakpoint you can issue the `print` command to examine the value of any expression containing variables that are in scope.

Examining the Call Stack

While paused at `IntMath_gcd()`, issue the `where` command:

```
(gdb) where
```

In response, `gdb` displays a call stack trace. Reading the output from bottom to top gives you a trace from a specific line of the `main()` function, through specific lines of intermediate functions, to the about-to-be-executed line.

The `where` command is particularly useful when your program is crashing via a segmentation fault error at runtime. When that occurs, try to make the error occur within `gdb`. Then, after the program has crashed, issue the `where` command. Doing so will give you a good idea of which line of your code is causing the error.

Quitting `gdb`

Issue the `quit` command to quit `gdb`:

```
(gdb) quit
```

Then, as usual, type:

```
<Ctrl-x> <Ctrl-c>
```

to exit `emacs`.

Command Abbreviations

The most commonly used `gdb` commands have one-letter abbreviations (`r`, `b`, `c`, `n`, `s`, `p`). Also, pressing the Enter key without typing a command tells `gdb` to reissue the previous command.

Part 2: Reference

`gdb [-d sourcefiledir] [-d sourcefiledir] ... program [corefile]`
`ESC x gdb [-d sourcefiledir] [-d sourcefiledir] ... program`

Run `gdb` from a shell
Run `gdb` within Emacs

Miscellaneous	
<code>quit</code>	Exit <code>gdb</code> .
<code>directory [dir1] [dir2] ...</code>	Add directories <code>dir1</code> , <code>dir2</code> , ... to the list of directories searched for source files, or clear the directory list.
<code>help [cmd]</code>	Print a description of command <code>cmd</code> .

Running the Program	
<code>run [arg1],[arg2] ...</code>	Run the program with command-line arguments <code>arg1</code> , <code>arg2</code> , ...
<code>set args arg1 arg2 ...</code>	Set the program's command-line arguments to <code>arg1</code> , <code>arg2</code> , ...
<code>show args</code>	Print the program's command-line arguments.

Using Breakpoints	
<code>info breakpoints</code>	Print a list of all breakpoints.
<code>break [file:]linenum</code>	Set a breakpoint at line <code>linenum</code> in file <code>file</code> .
<code>break [file:]fn</code>	Set a breakpoint at the beginning of function <code>fn</code> in file <code>file</code> .
<code>condition bnum expr</code>	Break at breakpoint <code>bnum</code> only if expression <code>expr</code> is non-zero (TRUE).
<code>commands [bnum] cmds</code>	Execute commands <code>cmds</code> whenever breakpoint <code>bnum</code> is hit.
<code>continue</code>	Continue executing the program.
<code>kill</code>	Stop executing the program.
<code>delete [bnum1][,bnum2]...</code>	Delete breakpoints <code>bnum1</code> , <code>bnum2</code> , ..., or all breakpoints.
<code>clear [[file:]linenum]</code>	Clear the breakpoint at <code>linenum</code> in file <code>file</code> , or the current breakpoint.
<code>clear [[file:]fn]</code>	Clear the breakpoint at the beginning of function <code>fn</code> in file <code>file</code> , or the current breakpoint.
<code>disable [bnum1][,bnum2]...</code>	Disable breakpoints <code>bnum1</code> , <code>bnum2</code> , ..., or all breakpoints.
<code>enable [bnum1][,bnum2]...</code>	Enable breakpoints <code>bnum1</code> , <code>bnum2</code> , ..., or all breakpoints.

Stepping through the Program	
<code>next</code>	"Step over" the next line of the program.
<code>step</code>	"Step into" the next line of the program.
<code>finish</code>	"Step out" of the current function.

Examining Variables	
<code>print expr</code>	Print the value of expression <code>expr</code> .
<code>print [file'::]var</code>	Print the value of variable <code>var</code> as defined in file <code>file</code> . (<code>file</code> is used to resolve static variables.)
<code>print [function'::]var</code>	Print the value of variable <code>var</code> as defined in function <code>function</code> . (<code>function</code> is used to resolve static variables.)
<code>printf format, expr1, expr2, ...</code>	Print the values expressions <code>expr1</code> , <code>expr2</code> , ... using the specified <code>format</code> string.
<code>whatis var</code>	Print the type of variable <code>var</code> .
<code>pptype t</code>	Print the definition of type <code>t</code> .
<code>info display</code>	Print the display list.
<code>display expr</code>	At each break, print the value of expression <code>expr</code> .
<code>undisplay displaynum</code>	Remove <code>displaynum</code> from the display list.

Examining the Call Stack	
<code>where</code>	Print the call stack.
<code>frame</code>	Print the top of the call stack.
<code>up</code>	Move the context toward the bottom of the call stack.
<code>down</code>	Move the context toward the top of the call stack.

Working with Signals	
<code>info signals</code>	Print a list of all signals that the operating system makes available.
<code>handle sig action1 [action2 ...]</code>	When GDB receives signal <code>sig</code> , it should perform actions <code>action1</code> , <code>action2</code> , ... Valid actions are <code>nostop</code> , <code>stop</code> , <code>print</code> , <code>noprint</code> , <code>pass</code> , and <code>nopass</code> .
<code>signal sig</code>	Send the program signal <code>sig</code> .

Copyright © 2019 by Robert M. Dondero, Jr.