Princeton University
COS 217: Introduction to Programming Systems
Gdb Tutorial

This tutorial describes how to use a minimal subset of the gdb debugger. See the summary sheet distributed in precept for more information. Also see Chapter 6 of our *Programming with GNU Software* (Loukides & Oram) textbook.

The tutorial assumes that you have copied the testmymath (version 4) program, consisting of files mymath.h, mymath.c, and testmymath.c, from http://www.cs.princeton.edu/courses/archive/spr06/cos217/precepthandouts/04/mymath4/ to your working directory.

Introduction

Suppose you are developing the testmymath (version 4) program. Further suppose that the program preprocesses, compiles, assembles, and links cleanly, but is producing incorrect results at runtime. What can you do to debug the program?

One approach is temporarily to insert calls to printf(...) or fprintf(stderr, ...) throughout the code 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 code, step through your executing program one line at a time, examine the values of variables at breakpoints, examine the function call stack, etc.

Building for gdb

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

```
$ gcc -Wall -ansi -pedantic -g -o testmymath testmymath.c mymath.c
```

Doing so places extra information into the testmymath file that gdb uses.

Running gdb

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

```
$ xemacs
```

Now call the xemacs "gdb" function via these keystrokes:

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

At this point you are executing gdb from within xemacs. 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."

Using Breakpoints

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

```
(gdb) break main
(gdb) break MyMath_gcd
```

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 the about-to-be-executed line of code highlighted.

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 scanf(). Enter 8 as the first number. Execution is paused at the second scanf(). Enter 12 as the second number. Gdb is paused at the beginning of MyMath_gcd().

Then issue another "continue" command:

```
(gdb) continue
```

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

```
(gdb) break MyMath_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 MyMath_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 variable that is in scope.

Examining the Call Stack

While paused at MyMath_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 xemacs.

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.