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 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

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.

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> testintmath <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." Incidentally, file redirection is specified as part of the **run** command. For example, the command **run < somefile** runs the program, redirecting standard input to **somefile**.

**Using Breakpoints**

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

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

Incidentally, 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 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 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]  Run gdb from a shell
ESC x gdb [-d sourcefiledir] [-d sourcefiledir] ... program  Run gdb within Emacs

table

<table>
<thead>
<tr>
<th>Misellaneous</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>quit</td>
<td>Exit gdb.</td>
</tr>
<tr>
<td>directory [dir1] [dir2] ...</td>
<td>Add directories dir1, dir2, ... to the list of directories searched for source files, or clear the directory list.</td>
</tr>
<tr>
<td>help [cmd]</td>
<td>Print a description of command cmd.</td>
</tr>
</tbody>
</table>

table

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

table

<table>
<thead>
<tr>
<th>Using Breakpoints</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>info breakpoints</td>
<td>Print a list of all breakpoints.</td>
</tr>
<tr>
<td>break [file:]linenum</td>
<td>Set a breakpoint at line linenum in file file.</td>
</tr>
<tr>
<td>break [file:]fn</td>
<td>Set a breakpoint at the beginning of function fn in file file.</td>
</tr>
<tr>
<td>condition bpnum expr</td>
<td>Break at breakpoint bpnum only if expression expr is non-zero (TRUE).</td>
</tr>
<tr>
<td>commands [bpnum] cmd</td>
<td>Execute commands cmds whenever breakpoint bpnum is hit.</td>
</tr>
<tr>
<td>continue</td>
<td>Continue executing the program.</td>
</tr>
<tr>
<td>kill</td>
<td>Stop executing the program.</td>
</tr>
<tr>
<td>delete [bpnum1] [bpnum2] ...</td>
<td>Delete breakpoints bpnum1, bpnum2, ..., or all breakpoints.</td>
</tr>
<tr>
<td>clear [file:]linenum</td>
<td>Clear the breakpoint at linenum in file file, or the current breakpoint.</td>
</tr>
<tr>
<td>clear [file:]fn</td>
<td>Clear the breakpoint at the beginning of function fn in file file, or the current breakpoint.</td>
</tr>
<tr>
<td>disable [bpnum1] [bpnum2] ...</td>
<td>Disable breakpoints bpnum1, bpnum2, ..., or all breakpoints.</td>
</tr>
<tr>
<td>enable [bpnum1] [bpnum2] ...</td>
<td>Enable breakpoints bpnum1, bpnum2, ..., or all breakpoints.</td>
</tr>
</tbody>
</table>

table

<table>
<thead>
<tr>
<th>Stepping through the Program</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>&quot;Step over&quot; the next line of the program.</td>
</tr>
<tr>
<td>step</td>
<td>&quot;Step into&quot; the next line of the program.</td>
</tr>
<tr>
<td>finish</td>
<td>&quot;Step out&quot; of the current function.</td>
</tr>
</tbody>
</table>

table

<table>
<thead>
<tr>
<th>Examining Variables</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>print expr</td>
<td>Print the value of expression expr.</td>
</tr>
<tr>
<td>print [file:::]var</td>
<td>Print the value of variable var as defined in file file. (file is used to resolve static variables.)</td>
</tr>
<tr>
<td>print [function:::]var</td>
<td>Print the value of variable var as defined in function function. (Function is used to resolve static variables.)</td>
</tr>
<tr>
<td>printf format, expr1, expr2, ...</td>
<td>Print the values expressions expr1, expr2, ... using the specified format string.</td>
</tr>
<tr>
<td>whatis var</td>
<td>Print the type of variable var.</td>
</tr>
<tr>
<td>p type t</td>
<td>Print the definition of type t.</td>
</tr>
<tr>
<td>info display</td>
<td>Print the display list.</td>
</tr>
<tr>
<td>display expr</td>
<td>At each break, print the value of expression expr.</td>
</tr>
<tr>
<td>undisplay displaynum</td>
<td>Remove displaynum from the display list.</td>
</tr>
</tbody>
</table>

table

<table>
<thead>
<tr>
<th>Examining the Call Stack</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>where</td>
<td>Print the call stack.</td>
</tr>
<tr>
<td>frame</td>
<td>Print the top of the call stack.</td>
</tr>
<tr>
<td>up</td>
<td>Move the context toward the bottom of the call stack.</td>
</tr>
<tr>
<td>down</td>
<td>Move the context toward the top of the call stack.</td>
</tr>
</tbody>
</table>

table

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

Copyright © 2015 by Robert M. Dondero, Jr.