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

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

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

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

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

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

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

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