CS0449 — GDB Lab

What is a debugger?

A ***debugger*** is a program that helps you find logical mistakes in your programs by running them in a controlled way.

Undoubtedly by this point in your programming life, you've had a program or two that just didn't behave in the way you expected it to. Perhaps when this happened, you went and added some print statements to your program, recompiled it, and then ran it. Those print statements probably:

1. Allowed you to see the value of one or more variables, or
2. Indicated where you were in the execution of a program

or possibly both. A debugger is a more sophisticated tool that helps you do both of these tasks without having to edit your source code and recompile.

How does a debugger work?

The most important operation that a debugger provides is the ability to insert a ***breakpoint*** into a program. A breakpoint is a special trigger that, when program execution reaches a specified point, allows the debugger to pause your program and to take control.

With the program paused, you now can do various operations that query the state of the running program, such as print out the value of the variables or see a backtrace of the function calls that led to where the program paused.

What debugger will we use?

Many IDEs and programming environments provide some interface to a debugger, usually integrated with the code editor. On Linux however, we'll be using a command line debugger called gdb (the GNU debugger).

How does gdb work?

By default, the programs that we produce with gcc are not very conducive to being debugged. As part of the compilation process, things like most of our variable names, the name of the source file, and the line numbers therein are thrown away, since they serve no purpose to the CPU. To get around this problem, we can tell gcc to compile our program to better support being run under a debugger. We do this by specifying the -g option:

```
gcc -g -o executable_name source_code.c
```

Because of the extra information embedded into the program by gcc, most of the programs you will get from other people do not contain this extra debugging information. But as long as you have the source code, you can always produce a debug version yourself.

To invoke gdb on our executable, we simply say:

gdb executable_name

and on thoth, we'll be greeted with a message and then a prompt to type a command:

```
(37) thoth $ gdb executable_name
GNU gdb Red Hat Linux (6.3.0.0-1.162.el4rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...Using host libthread_db library
"/lib64/tls/libthread_db.so.1".

(gdb)
```

Commands in gdb

Here are some commonly used commands in gdb. You can learn more about all of the commands by typing help, or on a specific command by typing help command_name.

Command	Shortcut	Description
help		Get help on a command or topic
set args		Set command-line arguments
run	r	Run (or restart) a program
quit	q	Exit gdb
break	b	Place a breakpoint at a given location
continue	c	Continue running the program after hitting a breakpoint
backtrace	bt	Show the function call stack
next	n	Go to the next line of source code without entering a function call
step	s	Go to the next line of source code, possibly entering a new function
nexti	ni	Go to the next instruction without entering a function call
stepi	si	Go to the next instruction, possibly entering a new function
print		Display the value of an expression written in C notation
x		Examine the contents of a memory location (pointer)
list		List the source code of the program
disassemble	disas	List the machine code of the program

Example

Login to thoth.cs.pitt.edu via SSH. Change to your private directory (or your /u/SysLab directory if you're out of AFS space) and make a directory called gdb_lab. Go into that directory and issue the command:

cp ~aus4/public/cs449/gdbdemo.c .

(That period at the end represents the current directory, i.e., the destination of the copy.)Take a look at the contents of gdbdemo.c using pico or your favorite text editor. You might notice that there are some errors in the program that wouldn't be caught using the compiler only. Compile it and run it to see the result:

```
(23) thoth $ gcc -o gdbdemo gdbdemo.c
(24) thoth $ ./gdbdemo
Enter a number: 3
Segmentation fault
```

That means the program has crashed. Let's recompile with debugging support and see if gdb can't help us:

```
(25) thoth $ gcc -g -o gdbdemo gdbdemo.c
(26) thoth $ gdb gdbdemo
```

At the (gdb) prompt, we simply want to run the program, so type "run" or "r" and you'll be prompted to enter the number as if you had run the program normally. If you enter "3" as we did before, you'll see:

```
(gdb) r
Starting program: /afs/pitt.edu/home/a/u/aus4/private/gdb_lab/gdbdemo
Enter a number: 3

Program received signal SIGSEGV, Segmentation fault.
0x0000003ec1d5080f in _IO_vfscanf_internal () from /lib64/tls/libc.so.6
```

Gdb is telling us where the program crashed, but it is an odd function name. We see the "scanf" part and might assume that it's a problem with our input, but to be sure, let's find out by asking for a backtrace:

```
(gdb) back
#0  0x0000003ec1d5080f in _IO_vfscanf_internal () from /lib64/tls/libc.so.6
#1  0x0000003ec1d5884a in scanf () from /lib64/tls/libc.so.6
#2  0x0000000000400521 in main () at gdbdemo.c:11
```

This is the equivalent of a stack dump when we get an exception in Java. It turns out we were right, it's from our call to scanf, which occurs in main() at line 11 of gdbdemo.c (the colon introduces a line number). Let's find out what's at line 11 by using the list command:

```
(gdb) list gdbdemo.c:11
6       {
7               int x;
8               char c[30];
9
10               printf("Enter a number: ");
11               scanf("%d",x);
12
13               fun();
14
15               return 0;
```

Oh, how silly, I forgot the ampersand on the variable x. Quit gdb by issuing the "quit" command, edit the ampersand in (&x) and recompile the program. Run it.

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
(27) thoth $ pico gdbdemo.c
(28) thoth $ gcc -g -o gdbdemo gdbdemo.c
(29) thoth $ ./gdbdemo
Enter a number: 3
Floating point exception
```

Uh oh, it crashed again. Let's run it through gdb and see what happens:

```
(gdb) run
Starting program: /afs/pitt.edu/home/a/u/aus4/private/gdb_lab/gdbdemo
Enter a number: 3

Program received signal SIGFPE, Arithmetic exception.
0x000000000040055b in fun () at gdbdemo.c:25
25              c = a / b;
```

Hmm, it seems that it is crashing at line 25, with the expression c=a/b. Let's print out the values of these three variables to see why:

```
(gdb) print c
$1 = 0
(gdb) print a
$2 = 5
(gdb) print b
$3 = 0
```

Oh, I'm dividing by zero. I wonder how that is happening. I see that I'm in the function fun() since gdb told me that when it crashed. (I could also discover it via the backtrace command). I now would like to trace through the function to see how b gets set to zero. I can do this by placing a breakpoint at the function fun() so that execution stops whenever the function is called. I do this with the break command:

```
(gdb) break fun
Breakpoint 1 at 0x40053b: file gdbdemo.c, line 20.
```

I can also specify specific line numbers to stop at such as gdbdemo.c:20. We should restart the program by typing “run” again and enter our input when prompted:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /afs/pitt.edu/home/a/u/aus4/private/gdb_lab/gdbdemo
Enter a number: 3

Breakpoint 1, fun () at gdbdemo.c:20
20          int a = 5;
```

Our program has now paused before the statement at line 20 as displayed here. Let’s type “next” to go to the next line:

```
(gdb) next
21          int b = 0;
```

And there, we find the culprit, a simple initialization statement. We can allow the program to resume with the “continue” command and as we expect, it crashes:

```
(gdb) continue
Continuing.

Program received signal SIGFPE, Arithmetic exception.
0x000000000040055b in fun () at gdbdemo.c:25
25          c = a / b;
```

If we’d like, we now know where to fix our program to avoid this crash.

Recap

When we find a program with a bug, we can interactively debug it using a program like gdb. We needed to compile it specially with the -g option to get the extra debug information included in the executable. Once we did that, we could run the program through gdb. Gdb provides a variety of commands to help us determine the value of variables and the flow of execution. We examined only a few of the essential commands such as print, break, run, next, and continue.

For further study

In the class textbook available in PDF form online, there is an Appendix B devoted to having a better understanding of what gdb provides for us. Additionally, in the references, there is a link to the gdb manual. Be aware your project 2 focuses heavily on using gdb. Also, for all future problems that your programs might have, we expect you to spend some time and effort to debug them yourselves before you email or stop by office hours. We’re still happy to help you if you don’t understand what gdb is telling you.