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:

```plaintext
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 (GDB) Red Hat Enterprise Linux (7.2-56.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
(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**.

<table>
<thead>
<tr>
<th>Command</th>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td></td>
<td>Get help on a command or topic</td>
</tr>
<tr>
<td>set args</td>
<td></td>
<td>Set command-line arguments</td>
</tr>
<tr>
<td>run</td>
<td>r</td>
<td>Run (or restart) a program</td>
</tr>
<tr>
<td>quit</td>
<td>q</td>
<td>Exit gdb</td>
</tr>
<tr>
<td>break</td>
<td>b</td>
<td>Place a breakpoint at a given location</td>
</tr>
<tr>
<td>continue</td>
<td>c</td>
<td>Continue running the program after hitting a breakpoint</td>
</tr>
<tr>
<td>backtrace</td>
<td>bt, back</td>
<td>Show the function call stack</td>
</tr>
<tr>
<td>where</td>
<td></td>
<td>Same as <strong>backtrace</strong></td>
</tr>
<tr>
<td>next</td>
<td>n</td>
<td>Go to the next line of source code without entering a function call</td>
</tr>
<tr>
<td>step</td>
<td>s</td>
<td>Go to the next line of source code, possibly entering a new function</td>
</tr>
<tr>
<td>nexti</td>
<td>ni</td>
<td>Go to the next instruction without entering a function call</td>
</tr>
<tr>
<td>stepi</td>
<td>si</td>
<td>Go to the next instruction, possibly entering a new function</td>
</tr>
<tr>
<td>print</td>
<td></td>
<td>Display the value of an expression written in C notation</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>Examine the contents of a memory location (pointer)</td>
</tr>
<tr>
<td>list</td>
<td>l</td>
<td>List the source code of the program</td>
</tr>
<tr>
<td>disassemble</td>
<td>disas</td>
<td>List the machine code of the program</td>
</tr>
</tbody>
</table>
Example
Login to thoth.cs.pitt.edu via ssh or PuTTY. Change to your private directory and make a directory called gdb_lab.
Go into that directory and issue the command:

```bash
cp ~jfb42/public/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 (core dumped)
```

That means the program has crashed. Let’s recompile with debugging support and see if gdb can 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 then enter, 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/j/f/jfb42/private/gdb_lab/gdbdemo
Enter a number: 3
Program received signal SIGSEGV, Segmentation fault.  
0x00000003ec1d5080f in __IO_vfscanf_internal () from /lib64/tls/libc.so.6  
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.132.el6_5.3.x86_64
```

(You can ignore the "Missing separate debuginfos" message. It has to do with the fact that this program is using a shared library.)

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  0x00000003ec1d5080f in __IO_vfscanf_internal () from /lib64/tls/libc.so.6  
#1  0x00000003ec1d5884a in scanf () from /lib64/tls/libc.so.6  
#2  0x00000000000400521 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 number after the colon is the line number). Let’s find out what’s at line 11 by using the list command:
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
```

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

```
(gdb) run
Starting program: /afs/pitt.edu/home/j/f/jfb42/private/gdb_lab/gdbdemo
Enter a number: 3
Floating point exception
```

Hmm, it seems that it is crashing at line 25, with the expression c=a/b. One thing the debugging info includes is the names of local variables. 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 or b 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/j/f/jfb42/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 "n" to go to the next line:

(gdb) n
21              int b = 0;

And there, we find the culprit, a simple initialization statement. We can allow the program to resume with the "c" command (short for continue) and as we expect, it crashes:

(gdb) c
Continuing.

Program received signal SIGFPE, Arithmetic exception.
0x0000000000040055b 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.
Looking under the hood

If you weren't around in class on Thursday, follow these steps:

1. Open a file in pico like so: `pico ~/.gdbinit` (the ~ is shorthand for your home directory.)
2. In the file, type this: `set disassembly-flavor intel`
3. Save the file and exit.

This will use the far more readable Intel disassembly syntax in gdb. (Unless you prefer the AT&T syntax. In which case I ask: what is wrong with you?)

To get more comfortable with the way the program looks in machine code, try compiling it with the options:

```
gcc -g -m32 -o gdbdemo gdbdemo.c
```

The -m32 option will compile it as a 32-bit x86 executable. If you forget that, it'll make a 64-bit executable, and when you disassemble you'll see all kinds of weird registers like `rax` and `xmm0` and who knows what.

Set breakpoints on `main` and `fun`, and when the breakpoint is hit, use `disas /m` (there's a space between `disas` and `/m`) to see what the disassembly looks like. Notice what the division in `fun` turns into... kinda weird, right? It's because the `idiv` instruction operates implicitly on registers `eax` and `edx`. `edx` is treated as the high 32 bits of the number to be divided, and `eax` as the low 32 bits. These two registers are treated as a 64-bit pair called `edx:eax`. The stuff with `mov edx, eax` and `sar edx, 0x1f` is actually sign-extend the divisor into the upper 32 bits of that 64-bit pair. Finally, `idiv` divides that 64-bit number by its single operand. Very weird.

Lots of x86 instructions operate on registers implicitly like this. It's best to have a good instruction reference on hand, like [this official Intel document](#). Chapter 3 has a listing of all the hundreds of instructions in IA-32. You can ignore any references to `AX` or "segment selectors" or anything like that. Just focus on the stuff that talks about `EAX` and such.

For further study

In the class textbook available in PDF form online, there are two appendices. Appendix A goes into some more detail about x86. Appendix B is devoted to giving you a better understanding of what `gdb` provides for us. Additionally, in the references, there is a link to the `gdb` manual.

Be aware your new project will focus heavily on using `gdb` and that for all future problems that your programs have, we expect you to have spent 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.

What to turn in

Nnnnnnnnothing! Go work on project 2. 😊