Introduction to Debugging

Mike Korcha, webmaster@cse-club.com

Winter 2015

Introduction

In this workshop, we will be learning how to use `gdb`, the GNU Debugger, which allows us to better locate faults, see the values of variables that are being used, and learn specifically why an error is occurring. By the end of this workshop, you will be able to use `gdb` to:

- Use breakpoints to pause a program at a given point
- View and change the values in variables while the program is running
- View a program's stack trace when it crashes

Getting Set Up

To complete this workshop, you'll need the following:

- `g++` compiler
- `gdb`
- Some sort of text editor (gedit, vim, etc)

These have been available forever, so you shouldn't have any issues getting them if you don't have access to them already. Any recent-ish version should do.

Lets Write a ~~Bad~~ Fantastic Program!

How will be possibly be able to debug if we don't have a sample program to work with? For educational purposes,we'll write a program that will inevitably have an issue that we will have to figure out. For accuracy purposes, we'll call it `divider_that_wont_break.cpp`

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[]) {
    int iterations = 10;

    for(int i = 0, j = iterations / 2 - 1; i < iterations; i++, j--) {
        cout << i / j << endl;
    }

    return 0;
}
```

Do you see the problem yet? No? Let's try to compile and run it:

```
g++ divider_that_wont_break.cpp
./a.out
```

Now, let's look at our output:

```
0
0
1
3
Floating point exception (core dumped)
```

Uh oh! It seems we're having issues somewhere! How will we figure this out? Well, we do want to learn about this tool called gdb, so let's start there!

Preparing to Debug

As you may have learned by now, when you invoke g++ from the command line, you're able to pass different parameters to it to have certain actions applied to it, ranging from an output filename with the `-o <...>` flag to increased warning output with `-Wall`. To get our program to have debug symbols, there's another flag we can pass to it: `-ggdb`. This flag will make the program a bit larger, but will also include everything needed to properly use the debugger.

Let's try recompiling now with the new flag:

```
g++ divider_that_wont_break.cpp -ggdb
```

Now, we'll load up the debugger instead of running as we normally do. This way, we'll have access to the features that gdb can provide.

```
gdb a.out
```

You should see something similar to the following:

```
GNU gdb (GDB) 7.9
Copyright (C) 2015 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-unknown-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...done.
(gdb)
```

If so, you are now in the gdb prompt! To run the program, just type "run", and your program will run as usual.

```
(gdb) run
Starting program: /home/mike/gdb-workshop/a.out
0
```

```
0
1
3

Program received signal SIGFPE, Arithmetic exception.
0x000000000040084b in main (argc=1, argv=0x7fffffffe908) at divider_that_wont_break.cpp:10
10          cout << i / j << endl;
(gdb)
```

As you can see, it shows a bit more about what's going on inside your program - showing the Arithmetic exception we're running into. It also shows what line the problem happened and what specifically is on the line:

```
10          cout << i / j << endl;
```

Breakpoints

Breakpoints are arguably one of the most useful things to use when debugging, and are incredibly simple to use in gdb. When a breakpoint is reached, the program will pause execution at that point to allow further inspection on what is going on, such as examining variable values. For our program, let's create a breakpoint to stop at the line we may be crashing at:

```
break 10
```

This tells the debugger to make a breakpoint at line 10. You can also create a breakpoint based on a function name or filename. You can clear a breakpoint by using the same identifier you used to create it:

```
clear 10
```

At this point, let's try running our program again with the `run` command:

```
(gdb) break 10
Breakpoint 1 at 0x400847: file divider_that_wont_break.cpp, line 10.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/mike/gdb-workshop/a.out

Breakpoint 1, main (argc=1, argv=0x7fffffffe908) at divider_that_wont_break.cpp:10
10          cout << i / j << endl;
(gdb)
```

You'll notice you've paused at this point in execution. To continue running, simply use the `step` command:

```
(gdb) step
0
9       for(int i = 0, j = iterations / 2 - 1; i < iterations; i++, j--) {
(gdb) step

Breakpoint 1, main (argc=1, argv=0x7fffffffe908) at divider_that_wont_break.cpp:10
10          cout << i / j << endl;
(gdb) step
```

```
0
9       for(int i = 0, j = iterations / 2 - 1; i < iterations; i++, j--) {
(gdb) step

Breakpoint 1, main (argc=1, argv=0x7fffffffe908) at divider_that_wont_break.cpp:10
10          cout << i / j << endl;
(gdb) step
1
9       for(int i = 0, j = iterations / 2 - 1; i < iterations; i++, j--) {
(gdb) step

Breakpoint 1, main (argc=1, argv=0x7fffffffe908) at divider_that_wont_break.cpp:10
10          cout << i / j << endl;
(gdb) step
3
9       for(int i = 0, j = iterations / 2 - 1; i < iterations; i++, j--) {
(gdb) step

Breakpoint 1, main (argc=1, argv=0x7fffffffe908) at divider_that_wont_break.cpp:10
10          cout << i / j << endl;
(gdb) step

Program received signal SIGFPE, Arithmetic exception.
0x000000000040084b in main (argc=1, argv=0x7fffffffe908) at divider_that_wont_break.cpp:10
10          cout << i / j << endl;
(gdb) step

Program terminated with signal SIGFPE, Arithmetic exception.
The program no longer exists.
(gdb)
```

How can this help us? Let's run again, and this time observe the variables.

Observing Variables

When we're in a stopped state, we can take a peek at what's going on under the hood, and see what values our variables are holding. This is another key feature that debuggers offer, as you can see what's going on in real-time with the program.

Let's rerun the program, and at the first breakpoint we'll observe the variables of our loop using the `print` command:

```
(gdb) run
Starting program: /home/mike/gdb-workshop/a.out

Breakpoint 2, main (argc=1, argv=0x7fffffffe908) at divider_that_wont_break.cpp:10
10          cout << i / j << endl;
(gdb) print i
$1 = 0
(gdb) print j
$2 = 4
```

So far so good, nothing alarming happening. Let's continue doing this until we see the problem...

```
(gdb) step
```

```
Breakpoint 1, main (argc=1, argv=0x7fffffffe908) at divider_that_wont_break.cpp:10
10          cout << i / j << endl;
(gdb) print i
$11 = 4
(gdb) print j
$12 = 0
```

You'll reach something like this eventually. Notice the issue now? We're trying to divide by 0, also known as blowing up the universe. Our computer doesn't want to do that, so crashes the program. What happens if we want to see what would happen with a quick fix, such as changing j to 1? We can do that with the `set` command.

```
(gdb) set j=1
(gdb) step
4
9       for(int i = 0, j = iterations / 2 - 1; i < iterations; i++, j--) {
(gdb) step

Breakpoint 2, main (argc=1, argv=0x7fffffffe908) at divider_that_wont_break.cpp:10
10          cout << i / j << endl;
```

Now we're able to step through that iteration of the loop. While not fixing our program, we know exactly what our problem is, and can fix it in our code with ease:

```
if(j == 0) {
    cout << i / 1 << endl;
}
else {
    cout << i / j << endl;
}
```

Let's kill the program, quit the debugger, recompile, and run it again.

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) quit

g++ divider_that_wont_break.cpp -ggdb
./a.out
```

If you see something similar to this:

```
0
0
1
3
4
-5
-3
-2
-2
-1
```

Then you have successfully fixed the program.

Let's Write Another ~~Horrendous~~ Perfect Program!

Let's play around with one more example. This time, we'll have a program that we're gonna keep running in the background, mindlessly calculating the slopes of two random points. Call it `forever_a_slope.cpp`.

```
#define ever (;;)

#include <iostream>
#include <cstdlib>
#include <ctime>
#include <unistd.h>

using namespace std;

int randint(int min, int max) {
    return rand() % max + min;
}

int main(int argc, char* argv[]) {
    srand(time(NULL));

    int slope = 0;

    for ever {
        slope = (randint(1, 10) + randint(1, 10)) / (randint(1, 10) + randint(1, 10));

        sleep(1);
    }

    return 0;
}
```

We'll compile it with our debug flag as we've been doing and then run it in the background using an '&' at the end of the command.

```
g++ forever_a_slope.cpp -ggdb
./a.out &
```

If you see something similar to the following, then you now have a process running in the background:

```
[mike@korcha-arch gdb-workshop]$ ./a.out &
[1] 2617
```

The number shown is the process ID of the program you just ran, so you can kill it later if need be. We'll need this when we attach to the process.

Attaching to a Running Process

Now, we'll invoke `gdb` with a flag and the process ID we just got as the argument for the program. It will then hook in wherever the program is at in execution. Note that this does require root, and as such is only doable on machines you have full control over.

```
gdb --pid 2617

GNU gdb (GDB) 7.9
Copyright (C) 2015 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-unknown-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
Attaching to process 2617
Reading symbols from /home/mike/gdb-workshop/a.out...done.
Reading symbols from /usr/lib/libstdc++.so.6...done.
Reading symbols from /usr/lib/libm.so.6...(no debugging symbols found)...done.
Reading symbols from /usr/lib/libgcc_s.so.1...done.
Reading symbols from /usr/lib/libc.so.6...(no debugging symbols found)...done.
Reading symbols from /lib64/ld-linux-x86-64.so.2...(no debugging symbols found)...done.
0x00007fe39ef43e90 in __nanosleep_nocancel () from /usr/lib/libc.so.6
(gdb)
```

Lets take a look at the stack trace to see where the program is currently executing (most likely in `sleep()`). The stack trace is a way to see what's going on in the chain of execution on the program stack. We can view it using either `bt` or `backtrace`

```
(gdb) bt
#0  0x00007fe39ef43e90 in __nanosleep_nocancel () from /usr/lib/libc.so.6
#1  0x00007fe39ef43d44 in sleep () from /usr/lib/libc.so.6
#2  0x0000000000400867 in main (argc=1, argv=0x7fff54cfc168) at forever_a_slope.cpp:23
(gdb)
```

We can also do anything we were doing previously with a program we started through `gdb`, such as printing values.

Closing Remarks

This workshop was an introduction to using debug tools, `gdb` in particular. Various IDEs incorporate or provide their own, and other compilers may use other tools, but the concepts behind them are relatively similar. These techniques aren't limited to just the scenarios covered here - you can easily use these to find where segmentation faults are occurring, where you may be using the STL wrong, incorrect pointer arithmentic, and more.

Hopefully this introduction will prove useful in future coursework and in other projects.

Note - The programs created in this were created with no purpose other than to demonstrate debugging in this workshop. They (probably) have no practical use otherwise.