A GDB Tutorial with Examples

By Manasij Mukherjee

A good debugger is one of the most important tools in a programmer's toolkit. On a UNIX or Linux system, GDB (the GNU debugger) is a powerful and popular debugging tool; it lets you do whatever you like with your program running under GDB.

Should you read this?

You should... if you can relate to two or more of the following:

- You have a general idea of programming with C or C++.
- You put a lot of cout or printf statements in the code if something goes wrong.
- You have used a debugger with an IDE, and are curious about how the command line works.
- You've just moved to a Unix-like operating system and would like to know about the toolchain better.

A crash course on compiling with gcc (or g++)

Gcc is the de facto compiler in Linux or any other *nix system. It also has Windows ports but on Windows, you'll probably find the debugger in Visual Studio 'easier'.

Suppose you have a file called main.cpp containing your c++ code. You should compile it with the following command:

```
g++ main.cpp -o main
```

While this will work fine and produce an executable file called main, you also need to put a -g flag to tell the compiler that you may want to debug your program later.

So the final command turns into:

```
g++ main.cpp -g -Wall -Werror -o main
```

(If you're wondering what -Wall and -Werror are, you may find this page on GCC a useful read.)

Don't worry if it looks cumbersome, you'll get used to it! (If you've got multiple source files you should use a good build system like make or Scons.)

The Basics of GDB

Provided you've compiled your program with the debugging symbols enabled, you're ready to start debugging. Any time there is text you should replace, I've put it in <angle brackets>.

Starting GDB

To start GDB, in the terminal,

```
gdb <executable name>
```

For the above example with a program named main, the command becomes

```
gdb main
```

Setting Breakpoints

You'll probably want you program to stop at some point so that you can review the condition of your program. The line at which you want the program to temporarily stop is called the breakpoint.

```
break <source code line number>
```

Running your program

To run your program, the command is, as you guessed,

```
run
```

Looking at the code

When the program is stopped, you can do a number of important things, but most importantly you need to see which part of the code you've stopped. The command for this purpose is "list". It shows you the neighbouring 10 lines of code.

Next and Step

Just starting and stopping isn't much of a control. GDB also lets you to run the program line-by-line by the commands 'next' and 'step'. There is a little difference between the two, though. Next keeps the control strictly in the current scope whereas step follows the execution through function calls.

Look at this example carefully;

Suppose you have a line in the code like

```
value=display();
readinput();
```

If you use the next command, the line (and the function, provided there aren't breakpoints in it) gets executed and the control advances to the next line, readinput(), where you can perhaps examine 'value' to get an idea of how display() worked.

But if you use the step command, you get to follow what display() does directly, and the control advances to the first line of display(), wherever it is.

Examining your Variables

When you want to find the misbehaving portion of your program, it often helps to examine local variables to see if anything unexpected has occurred. To examine a variable, just use

```
print <var name to print>
```

Note: You can also modify variables' values by

```
set <var> = <value>
```

You can modify variables to see if an issue is resolved if the variable has another value or to force the program to follow a particular path to see if the reason for a bug was due to a variable having the wrong value.

Setting Watchpoints

Setting watchpoints is like asking the debugger to provide you with a running commentary of any changes that happen to the variables. Whenever a change occurs, the program pauses and provides you with the details of the change.

The command to set a simple watchpoint (a write watchpoint, i.e you are notified when the value is written) is

```
watch <var>
```

Here's some example output when GDB pauses due to a change in <var>:

```
Continuing.
Hardware watchpoint 2: variable

Old value = 0
New value = 1
0x08048754 at main.cpp:31
31         variable=isalpha(ch)
```

Note: You can only set watchpoints for a variable when it is in scope. So, to watch something within another function or a inner block, first set a breakpoint inside that scope and then when the program pauses there, set the watchpoint.

Quit

To stop your program, when it is paused, use kill and to quit GDB itself, use quit.

An Example Debugging Session

The given code computes the factorial of a number erroneously. The goal of the debugging session is to pinpoint the reason of the error.

```
 1  #include<iostream>
 2
 3  using namespace std;
 4
 5  long factorial(int n);
 6
 7  int main()
 8  {
 9      int n(0);
10      cin>>n;
11      long val=factorial(n);
12      cout<<val;
13      cin.get();
14      return 0;
15  }
16
17  long factorial(int n)
18  {
19      long result(1);
20      while(n--)
21      {
22          result*=n;
23      }
24      return result;
25  }
```

Into the Debugger

Now follow the commands and the outputs carefully, especially the watchpoints. What I'm doing is basically:

- Setting a breakpoint just in the line of the function call
- Stepping into the function from that line
- Setting watchpoints for both the result of the calculation and the input number as it changes.
- Finally, analyzing the results from the watchpoints to find problematic behaviour

```
$ g++ main.cpp -g -Wall -o main
$ gdb main
GNU gdb (GDB) Fedora (7.3-41.fc15)
Copyright (C) 2011 Free Software Foundation, Inc.
This GDB was configured as "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/manasij7479/Documents/main...done.
(gdb) break 11
Breakpoint 1 at 0x80485f9: file main.cpp, line 11.
```

```
(gdb) run
Starting program: /home/manasij7479/Documents/main
3

Breakpoint 1, main () at main.cpp:11
11          long val=factorial(n);
(gdb) step
factorial (n=3) at main.cpp:19
19          long result(1);
(gdb) list
14          return 0;
15        }
16
17        long factorial(int n)
18        {
19          long result(1);
20          while(n--)
21          {
22            result*=n;
23          }
(gdb) watch n
Hardware watchpoint 2: n
(gdb) watch result
Hardware watchpoint 3: result
(gdb) continue
Continuing.
Hardware watchpoint 3: result

Old value = 0
New value = 1
```

Notice that result starts from 0 and is initialized to 1.

```
factorial (n=3) at main.cpp:20
20          while(n--)
(gdb)
```

Notice that I didn't put in a command, I just hit <return>. It re-executes the last command.

```
Continuing.
Hardware watchpoint 2: n

Old value = 3
New value = 2
```

Notice that n gets is immediately decremented from 3 to 2.

```
0x08048654 in factorial (n=2) at main.cpp:20
20          while(n--)
(gdb)
Continuing.
Hardware watchpoint 3: result

Old value = 1
New value = 2
```

Now result becomes 2 (by multiplying result's earlier value with n's value). We've found the first bug! result is supposed to be evaluated by multiplying 3 * 2 * 1 but here the multiplication starts from 2. To correct it, we have to change the loop a bit, but before that, lets see if the rest of the calculation is correct.

```
factorial (n=2) at main.cpp:20
```

```
20          while(n--)
(gdb)
Continuing.
Hardware watchpoint 2: n

Old value = 2
New value = 1
```

n gets decremented from 2 to 1. Result doesn't change since n is 1.

```
0x08048654 in factorial (n=1) at main.cpp:20
20          while(n--)
(gdb)
Continuing.
Hardware watchpoint 2: n

Old value = 1
New value = 0
```

n gets decremented from 1 to 0.

```
0x08048654 in factorial (n=0) at main.cpp:20
20          while(n--)
(gdb)
Continuing.
Hardware watchpoint 3: result

Old value = 2
New value = 0
```

Now result becomes 0 (by multiplying result's earlier value with n's value, 0). Another bug! How can result hold the value of the factorial when it is multiplied by 0? The loop must be stopped before n reaches 0.

```
factorial (n=0) at main.cpp:20
20          while(n--)
(gdb)
Continuing.
Hardware watchpoint 2: n

Old value = 0
New value = -1
0x08048654 in factorial (n=-1) at main.cpp:20
20          while(n--)
(gdb)
Continuing.
```

Now n becomes -1 and the loop isn't permitted to run anymore because n-- returns 0, and the function returns result's current value 0. Let's see what happens when the function exits.

```

Watchpoint 2 deleted because the program has left the block in
which its expression is valid.

Watchpoint 3 deleted because the program has left the block in
which its expression is valid.
```

This is what happens to a watchpoint when the variable goes out of scope.

```
0x08048605 in main () at main.cpp:11
```

```
11          long val=factorial(n);
(gdb) print val
$1 = 1293357044
```

print val shows a garbage value because gdb points to a line before it is executed, not after.

```
(gdb) next
12          cout<<val;
(gdb) continue
Continuing.
0[Inferior 1 (process 2499) exited normally]
(gdb) quit
```

Here's what the fix should look like:

```
while(n>0) //doesn't let n reach 0
{
  result*=n;
  n--;        //decrements only after the evaluation
}
```

GDB in Conclusion

You have now seen enough to try GBD out on your own. Some important topics have not been touched upon here for the sake of simplicity, such as dealing with segmentation faults and other kinds of crashes or using tools like Valgrind to find memory leaks.

Remember that GDB comes built in with an excellent help system. Just type help in the (gdb) prompt and you will be presented with options of what you could need help with. For details about a specific command, use the syntax

```
help <command>
```

Another important point to note is the use of shortcuts (like 'q' for 'quit'). GDB lets you use shortcuts for commands when it is not ambigious.

After learning about GDB, you do not have to panic the next time your program goes crazy. You have an excellent weapon in your arsenal now.

Read more...

Advanced GDB Debugging

Learn about Valgrind, a memory leak detector and memory error hunter

More information about finding and debugging segmentation faults (segfaults)

Debugging with Visual Studio