Debugging with GDB

By Alexandra Hoffer

GDB (Gnu Debugger) is a powerful debugging tool for both C and C++. A debugger is a program that helps you find bugs by letting you examine the state of the program as it's running. You can view the code, see the values of the variables, follow the control flow, break in the middle of an infinite loop, and more. This document is a quick introduction to the most common and useful GDB commands and techniques.

Starting Out

To gdb your program, just type *gdb <program>* at the prompt. You can also start up gdb first and then if you have a core dump you want to examine, you can *load <program>*. Once you've done this, you can start running your program by typing *run*. You will then interact normally with your program until typing control-C, which will pop you back to GDB's interface. Type *continue* to move back to your program from GDB. All commands in GDB can be shortened to their first unique letters, so you can type *c* instead of *continue*. If you ever get confused, GDB has a very good built-in help system that shows you everything you can do with GDB and how to do it. Type *help* at the GDB prompt to access it. And once you've found your bug, type control-D to exit gdb. Running more than one instance of GDB at once can confuse it, so make sure you've only got one copy open at a time.

Looking Around

There are two main kinds of commands you can use at the GDB prompt. One kind of command tells GDB how to act; the other kind just tells it to give you information. In this section we'll discuss the latter kind of command. You can use GDB to tell you the values of variables by typing *print <variable>*. This kind of command works just as it would within C or C++; for example, you can look at a particular element in an array with *print myarray[5]* or a field in a structure with *print struct.label*.

List is another useful command to let you know where you are; it will show you the surrounding lines of code. The next time you type *list*, you will be shown the next 10 lines. You can look at a particular area of code by typing *list <line number>*. For example, if you want to look at the area of line 20, type *list 20*. If your program has multiple source files, you can specify which one you want to look at by typing *list<filename>:<line number>*. You can also choose to examine the beginning of a function using *list<function>*.

A third command, *backtrace*, is used for looking at the stack of your program. When you type *backtrace*, you will see a listing of all the functions that have not yet returned and where those functions were called. If you want to look at a frame above or below your current frame in the stack, use *up* and *down* to change your scope, so that you can see the state of the caller function.

Breaking In

There are many debugging techniques you can use with GDB, but there are a handful that are far more common and useful than the rest. One such technique is setting breakpoints. A breakpoint is a place where your code will automatically stop running so that you can examine the state of the program. To set a breakpoint, type *break <line number>*. *Break* uses the same syntax as *list*.

A command very useful in conjunction with *break* is called *next*. The *next* command will run the program for one line and then immediately bring you back to the GDB prompt. You can also give numerical arguments to *next*; *next 100*, for example, will make the program break after the 100th next instruction.

The command *watch* can also be handy--you tell the program to watch a variable and break whenever its value changes. However, setting watchpoints slows down the functioning of gdb enormously, so unless you have your own Unix box, you might want to avoid it.

Messing With

There is also another set of commands, although they are not very commonly used; these commands actually modify the program as it is running. The *call* command, for example, will call a particular function; the *set variable* command will set a variable in the program to a particular value, for example, *set variable i = 0*. *Return* will make the current function return to the caller.

Techniques

Now that you've learned all these neat commands, how do you use them? Here are a few common debugging problems and the techniques that go along with them:

Segfaults and Core Dumps

When your program segfaults and dumps its core, what to do? The simplest thing to do is to run the program in gdb and follow through until it segfaults. When your program crashes, GDB will step up and tell you what line it crashed on. Usually it will be a line where the program was trying to manipulate some memory-mangled variable. You can use *bt* (or *backtrace*) to figure out where you are in the stack, and

then `p` (or `print`) to look at the variables and see if any of them have stepped outside their bounds or are full of garbage. As a shortcut, you can gdb your program and then `load` the corefile to get directly to the crash.

Infinite Loops

Infinite loops are another bug that are hard to hunt down on your own, but simple to find with GDB. Load up your program in GDB, and run it until you get to the loop. Then hit control-C to force your way back to the GDB prompt. Type `bt` and you'll be told what line you're on. Use `list` to look at the code and see if you can figure out what's causing the loop. If you can't, use `n` (or `next`) to progress through the loop one line at a time, all the while using `p` to inspect your counter variable.

Inexplicable Errors

Most bugs aren't so simply categorized; often your program will provide the wrong output for no apparent reason. There are a few techniques that can come in handy when you have this sort of problem. One thing you can do is set a breakpoint right after your program asks for input, then `next` your way along, looking periodically at the stack and the variables to make sure that everything is going according to plan. If you think the problem is due to the value of a particular variable and your code is hard to follow, you can `watch` that variable.

Here follows a transcript of a GDB session with a program intended to parse regular expressions (this program was originally called rep.cc, but you shouldn't need the original source code to follow the debugging flow). My comments will be indicated in C++ comment style.

```
>gdb rep
GNU gdb 4.17
Copyright 1998 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 "alphaev56-dec-osf4.0d"...
(gdb) run
Starting program: /rep
Please enter a regular expression:
a*
The State Table:
0 a ---> 1
1 # ---> 2
2 # ---> 0
Please enter a string to test against that RE:

//Here I typed control-C

Program received signal SIGINT, Interrupt.
0x3ff800d5d68 in read ()

(gdb) bt

//The gdb prompt lets us know we're sending commands to GDB and not to the actual program we're testing.


#0 0x3ff800d5d68 in read ()
#1 0x3ff801362cc in __read_nc ()
#2 0x3ff800d9c38 in _filbuf ()
#3 0x12002c360 in stdiobuf::sys_read (this=0x0, buf=0x14000a3b3 "\n",
size=8192) at ../../../gcc-2.95.1/libio/stdiostream.cc:77

//The stuff at the top of the stack here is built-in C++ functions for reading input. Let's skip toward the bottom, where we'll see some of my
actual code:

#12 0x120018eb0 in main (argc=1, argv=0x11ffff868) at rep.cc:43

//GDB is telling us where the caller function was when it called up the next stack frame, and what arguments the caller had.

(gdb) list rep.cc:43

//GDB now shows us some code from our program

38 our_parser.check_parenthesis_matching();
39 cout << the_state_table;
40 do
41 {
42 cout << "Please enter a string to test against that RE:\n";
43 getline(cin, tested);
44 cout << "The string '" << tested << "' is ";
45 //the actual checking work
```

46 if (!(total_ndfa.run(tested)))
47 {

(gdb) c
Continuing.

//Because we stopped the program when it was asking for input, it will keep waiting for that input until we enter something.

a
The string 'a' is accepted by the RE "a*"
Do you want to test another string? [y/n]

n
Do you want to enter a different regular expression? [y/n]
n
Thanks for playing!

//At this point, GDB kicks in because the program has ended.
Program exited normally.
(gdb)

//Now let's show off some other features of GDB, like variable inspection and breakpoints.

(gdb) run
Starting program: /rep
Please enter a regular expression:
b
The State Table:
0 b ---> 1
1
Please enter a string to test against that RE:

//Here we type control-C to get to the GDB prompt

Program received signal SIGINT, Interrupt.
0x3ff800d5d68 in read ()
(gdb) break rep::yesno
Breakpoint 2 at 0x1200194e8: file rep.cc, line 68.

//We have just set a breakpoint
(gdb) c
Continuing.
a
The string 'a' is not accepted by the RE "b"
Do you want to test another string? [y/n]
Breakpoint 1, rep::yesno () at rep.cc:67

//We have reached the breakpoint we set, so the program halts. Note that the line numbers can vary slightly--we set the break at 68 but are actually breaking at 67--because of lines that don't contain any actual instructions.
67 {

(gdb) n

//This n is being read as "next", not as "no", because we're in gdb and not the program.

rep::yesno () at rep.cc:68
68 string answer;

//We break again because of typing "next"

(gdb) p answer
$1 = {static npos = 18446744073709551615, static nilRep = {len = 0, res = 0,
ref = 1, selfish = false}, dat = 0x12001969c ""}

//answer is full of garbage, because it hasn't been initialized, just declared.

(gdb) n
69 getline(cin, answer);
(gdb) n
n

```
//The first of those two n's was a "next" that lead us to a prompt where the program asked for input. The second n, however, was a "no"
70 if (answer[0] == 'Y' || answer[0] == 'y')

//The program read in the input and broke on line 70 because we had entered "next"
(gdb) n
74 return false;
(gdb) p answer
$2 = {static npos = 18446744073709551615, static nilRep = {len = 0, res = 0,
ref = 1, selfish = false}, dat = 0x1400148b0 "n"}

//Now answer, as we see, contains the string "n"

(gdb) bt
#0 rep::yesno () at rep.cc:74
#1 0x1200196a8 in rep::more_strings () at rep.cc:82
#2 0x120018fe4 in main (argc=1, argv=0x11ffff868) at rep.cc:51

(gdb) up
#1 0x1200196a8 in rep::more_strings () at rep.cc:82
82 return yesno();

//Now we are at the caller of yesno() because we moved to a different stack frame when we went up.

(gdb) p answer
No symbol "answer" in current context.

//Answer is out of scope now, so we can't examine it.
```

As you can see, GDB is a multifaceted tool. You can learn more about it at the GNU GDB site.

Related articles

GDB by Example

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

More information about finding and debugging segmentation faults (segfaults)

Debugging with Visual Studio