A guide to GDB

Menaka Lashitha Bandara

Monash University (Clayton),
Victoria, Australia.

lashi@optusnet.com.au

The latest copy of this document is available at:

`http://members.optusnet.com.au/lashi/academic_technical.html`

Contents

Introduction

Despite the existence of easily provable programming paradigms (such as functional or logical), imperative and object-oriented programming, have emerged to be the popular standard.

It's for this reason that Debuggers are needed. One is ridiculed with low-level nonsense when programming in an imperative and/or object-oriented language - although one's code might compile, there is no guarantee that the code will behave the way in which it was intended. Even the Überprogrammers make implementation-level mistakes in their code.

The Debugger is an application in which the programmer can step through their code, line by line, check status of variables that are within scope, and perform similar debugging operations. This takes a huge amount of weight off the programmers chest. If it weren't for the debugger, the programmer would have to spend many hours reading through code, and tracing the execution, attempting to find why their application causes a segmentation violation or enters an infinite loop.

GDB is the *GNU DeBugger*[1]. It is used to debug code that has been compiled by GCC (the GNU Compiler Collection). It's a very powerful debugger that allows you to debug even the most sophisticated of software. This document is intended for the beginner or advanced programmer, not as a comprehensive guide, but rather as an introduction to `gdb`. This guide assumes that the reader is familiar with the C language to a reasonable depth.

[1] www.gnu.org/software/gdb

Chapter 1

GDB Basics

1.1 Behind the scenes

Not every executable binary can be simply loaded into `gdb` for debugging. This is because it does not contain *debugging symbols*. These symbols more or less tells `gdb` where to look in the source when it's running a program.

Two things need to happen for `gdb` to be able to debug code. Firstly, you need to have the source tree in the same state as it was when you compiled your program. So, for example, if all the source files were compiled in `/home/user/program/src/`, and the binary was compiled in `/home/user/program/ui/`, then you can't move the sources elsewhere and expect gdb to "magically" know this.

The second thing is that you need to tell `gcc` to "implant" the symbols into the binary. You do this by specifying the `ggdb3` flag during compilation. So, for instance, compiling a `binary.c` to `binary`:

```
bash$ gcc -ggdb3 -o binary binary.c
```

It is worthwhile noting here that the `ggdb3` flag produces slower code. It should only be used when debugging.

Note here, that `bash$` is not a command, but the command line prompt.

1.2 Starting `gdb`

The GNU Debugger can be started in many ways. The easiest way is:

```
bash$ gdb binary
```

This automatically loads the executable binary file `binary` and `gdb` is ready to take user input.

When `gdb` is started, it should look somewhat like this:

```
GNU gdb 2002-08-18-cvs
Copyright 2002 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 "powerpc-linux".
(gdb)
```

If `gdb` was started without the executable binary as a parameter, then it can be explicitly loaded after `gdb` has started. This is done by:

```
(gdb) file binary
```

1.3 Environment and Features

Like with most GNU Software, `gdb` is implemented with tab completion. This more specifically means, that you can press tab on a command, and if there is a match it'll fill it for you, or give you a list of commands that match the substring you typed.

Pressing Control-C returns control back to the `gdb` shell. To quit `gdb`:

```
(gdb) quit
```

If you are in the middle of a session, `gdb` remembers the history of your typing. One of the nice things about this is that you can press return without a command, and it'll automatically execute the previous command. This is especially handy when you're stepping through code.

1.4 Listing source

Once you're in `gdb` , you can list the lines of your source-code, in order to navigate through your program. To do this, use the `list` command, or shorthand, `l`:

```
(gdb) list
3       /* GDB Guide snippets */
4
5       #include <stdio.h>
6
7       int
8       main (void)
9       {
10              int     i;
11
12              for ( i = 0; i < 2; i++ )
```

It is important to understand the output that has been shown above. On the left hand side, `gdb` outputs the line number. Next to it, it shows you the line of code. Generally, debugging commands are issued to `gdb` by reference to line number.

You can also tell `list` to print a function or method of a class, ie `list bogus_function` will list `bogus_function()`, or `list BogusClass::bogus_function` will list `bogus_function()` which is contained in the namespace of the class `BogusClass`.

1.5 Running

Once a symbol-tipped binary is loaded into `gdb` , you need to explicitly tell `gdb` to execute it. You do so by the use of the command `run`. Here's an example:

```
(gdb) run
Starting program: /home/lashi/University/csse_club/gdb/excercises/binary
Hello GDB!
Hello GDB!

Program exited normally.
```

You can also pass command line parameters by sitting them next to the `run` command. If you have already specified parameters, then a stand-alone `run` will automatically load your previously used command line arguments.

1.6 Printing

When debugging an application, it is important that you can see the status of instantiated objects (ie variables). This can be done by using the `print` command. The objects that you wish to see can be dereferenced and casted as you would normally in C or C++.

In this snippet of code, we have `int x`, `int *x_ref = &x`, and `int **x_double_ref = &x_ref`. So, here's an example of using `print`:

```
(gdb) print x
$4 = 1
(gdb) print *x_ref
$5 = 1
(gdb) print x
$6 = 1
(gdb) print x_ref
$7 = (int *) 0x7ffffad8
(gdb) print *x_ref
$8 = 1
(gdb) print x_double_ref
$9 = (int **) 0x7ffffadc
(gdb) print *x_double_ref
$10 = (int *) 0x7ffffad8
(gdb) print **x_double_ref
$11 = 1
(gdb) print (char) x
$12 = 1 '\001'
(gdb) print $12
$13 = 1 '\001'
(gdb)
```

Note the variables that start with `$`. These are temporary variables that `gdb` creates. Especially note the last command, you can directory refer to one of these `$` variables.

From the state of `$7`, `$9`, and `$10`, you can see that `gdb` is actually printing out the *memory addresses*. This is rightly so, because we're printing out the raw contents of pointers.

1.7 Help

You can get online help for `gdb` at any point, by simply typing the `help` command. This would give you a list of topics. Furthermore, you can type `help command`, where `command` is a `gdb` command, and it will give you information about that command.

Chapter 2

Breakpoints, Runtime navigation, and Conditions

2.1 Breakpoints

Breakpoints are probably the first thing you need to learn when using any debugging tool. When you define a breakpoint in some part of your program, `gdb` returns control back to you when it reaches that exact point. This is a very powerful feature, because you can stop the program at points where you suspect something might be happening.

Breakpoints are created using the `break` command. For instance, this is telling `gdb` to pass back control on line 19 and 21:

```
(gdb) break 19
Breakpoint 1 at 0x100004b8: file printing.c, line 19.
(gdb) break 21
Breakpoint 2 at 0x100004d8: file printing.c, line 21.
```

Once control is given back to you, the programmer, you can then use other commands to examine the state of your program. For instance, you can use `print` to examine the state of the variables currently in your scope.

Notice here that the number of the breakpoint is displayed. It's extremely important to note that each breakpoint has a unique number, because some commands we discuss later need to use that number to address that breakpoint.

You may also require to remove breakpoints from your debugging session. This can be done on per-breakpoint basis, or all at once via the command `delete` or `del`.

Calling `delete` with a numerical argument, causes the breakpoint associated with that numerical argument to be cleared. All breakpoints can be cleared by issuing `delete` without any arguments.

2.2 Runtime Navigation

When you've been given control by `gdb` after it has reached a breakpoint, there are two commands that allow you to navigate through the sources. These are particularly useful for debugging iterative routines,

and nested function calls.

The first of these commands is `next` or `n`. Next should be used when you simply want to remain in the scope of the current function. So, if you've reached a function call, using `next` will cause that function to be executed before control is returned to you. Example:

```
Breakpoint 1, main () at stepnext.c:16
16                      function_call (i);
(gdb) next
0
15              for ( i = 0; i < 10; i++ )
(gdb)
```

The second command is `step` or `s`. Using `step`, allows you to get more into the detail of the program. It basically steps into a function call. For instance:

```
Breakpoint 1, main () at stepnext.c:16
16                      function_call (i);
(gdb) step
function_call (parameter=2) at stepnext.c:25
25              printf ("%d\n", parameter);
(gdb)
```

Just because `step` allows you to see the finer grains of code, it doesn't necessarily mean it's better. For instance, stepping into a function (like `printf()`) that doesn't have any debugging symbols causes `gdb` to complain that until control returns the end of that function. The best thing to do here is keep pressing `next` (it does happen quite often).

The third and last control command is `continue` or `c`. This command basically goes over all the code until you hit the next break point. This command is useful in loops and when you have large programs with many breakpoints, and you want to travel from one to another quickly. Here's what happens when we `continue` with the example we've been using:

```
(gdb) continue
Continuing.
7

Breakpoint 1, main () at stepnext.c:16
16                      function_call (i);
(gdb)
```

2.3 Conditions

Conditions are one of the most useful features of `gdb`. They return control back to you, when the condition is met for a pre-defined breakpoint. Conditions are defined by using the command `condition`, and can only be placed on pre-defined breakpoints (ie you need to firstly use `break`, then place a condition on that breakpoint - which you can address by the number).

The reason that conditions are not separate from breakpoints is because of scoping. There might by ten variables in your code identified by `i`. So, if you put a condition on the variable `i`, `gdb` won't know which `i` you're after.

For an example, lets say that hypothetically you're trying to debug a loop. And in this loop, you should never reach a certain state in a variable. You can place a condition on this variable and unlike the normal

behaviour of a breakpoint, control will only return to you when the condition is met. This simplifies the tiresome process of otherwise having to step through the code and print the status of this variable each time until you see it reach that condition.

Here's an example:

```
(gdb) break 16
Breakpoint 1 at 0x100004ac: file stepnext.c, line 16.
(gdb) condition 1 i == 5
(gdb) run
Starting program: /home/lashi/University/csse_club/gdb/excercises/stepnext
0
1
2
3
4

Breakpoint 1, main () at stepnext.c:16
16                      function_call (i);
(gdb) print i
$1 = 5
(gdb)
```

Note what's happened here. A breakpoint (breakpoint 1) has been defined at line 16, and a condition has been defined for breakpoint 1, namely `i == 5`. We see that control has been returned to us when, and only when `i == 5`.

Chapter 3

Core dumps, Back Tracing, and Stack Frames

3.1 What are core dumps

A core dump occurs when a program is abnormally terminated because it tried to do something illegal. For instance, if a process tried to divide by zero, which is an operation that's not defined, then it causes an interrupt on the machine, and an operating system like Linux takes the stack/context of that process, and dumps it into a file.

This is very useful, for a number of reasons. Firstly, not everyone knows how to program. Secondly, even those who are Überprogrammers may not have the time to debug ten thousand lines of someone else's code, because that would mean learning data structures, class hierarchies, API's and all that jazz. Thirdly, it'll be awfully slow if the binaries you ship out into real-life use are all tipped with debugging symbols. Generally code is optimised, and the only thing that they have in common are the context of variables in existence (and this is stretching the truth).

However, the most important reason is that it might be a rare, unreplicable bug. Bugs don't always reveal their presence in a deterministic way (because people give input, and people are not always deterministic). Thus, a bug may only manifest itself in rare and unique circumstances. But that's not an excuse against eliminating the bug.

The GNU Debugger can take a core dump of a program, along with a copy of that program that's been compiled with `ggdb3`, and debug the program from the core.

A core can be loaded into `gdb` in two ways. You can either load the core with the binary (note that `structbug` is the name of the binary):

```
bash$ gdb structbug core
```

Otherwise:

```
(gdb) target core core
```

where the last argument is the name of the core file (generally `core`).

Using the first method, `gdb` should look something like this:

```
Core was generated by `./structbug'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld.so.1...done.
Loaded symbols for /lib/ld.so.1
#0  0x10000528 in work_function (b=0x0) at structbug.c:44
44              b->x = 0;
(gdb)
```

3.2 Back Tracing

When you have a core file loaded, you can check the status of your application when the core was dumped. This is called a back trace. In `gdb` , you can issue `backtrace` or `bt`. For instance:

```
#0  0x10000528 in work_function (b=0x0) at structbug.c:44
#1  0x100004a4 in main () at structbug.c:26
#2  0x0fecac50 in __libc_start_main () from /lib/libc.so.6
(gdb)
```

The numbers after the hashes tell us the *Stack Frames*. Each function has it's own scope, and thus, it has a separate stack frame. The smallest number, ie zero, tells us where directly the segmentation violation occurred, and the higher numbers tell us where it was called from.

Note that back tracing is not only limited to core files. You could load any `ggdb3` tipped binary executable, and simply run it without any breakpoints, and after it abnormally terminates, issue a back trace.

3.3 Stack Frames

After we have obtained the numbers of the stack frames, we can use `frame` command to select these frames.

From these frames, we can analyse the state of all the variable within the scope of that frame when the segmentation violation occurred. This is very useful information when debugging code.

So, we'll look at the problem from inside to out. So, looking into the initial frame:

```
(gdb) frame 0
#0  0x10000528 in work_function (b=0x0) at structbug.c:44
44              b->x = 0;
(gdb) print b
$3 = (struct _BogusStruct *) 0x0
(gdb)
```

We see immediately that we're trying to access an element in a data structure which is not in existence. So, we'll move further outward and see what happened:

```
(gdb) frame 1
#1  0x100004a4 in main () at structbug.c:26
26              work_function (b);
```

```
(gdb) print b
$1 = (struct _BogusStruct *) 0x0
(gdb)
```

From this, we can tell that `work_function()` is not causing the problem. We can essentially look at the source now, before line 26, and see where the variable `b` was last modified.

References

Technical References

- *GDB Homepage.*
  www.gnu.org/software/gdb
- *GDB Documentation*
  www.gnu.org/software/gdb/documentation
- **Goerzen, John**. *Linux Programming Bible.*

Spiritual References

- **James, Geoffrey**.*The Tao of Programming.*
- **Stallman, Richard**.*GNU Philosophy.*
  www.gnu.org/philosophy