A Guide to Working with GDB

WWU CS Mentors

November 2, 2017

Contents

Chapter 1

Getting Started with GDB

When debugging code, the first instinct of most programmers is to use print statements to determine the value of variables and understand what the code is actually doing. However, there are powerful tools out there to aid in this process. As Patrick Star once said, "Wait Spongebob, we're not cavemen! We have technology."

1.1 Running GDB

GDB is the **G**NU Project **D**e**b**ugger. This tool works in tandem with compiled C programs and allows us to take a deeper look at the mechanisms of the code and follow along as it progresses. Let's begin with the sample code (test.c) found in Chapter 3. First, we need to compile the program with the -g command and invoke GDB:

```
user@cf162-01:~\$ gcc -g -o test test.c
user@cf162-01:~\$ gdb test
```

Once runnning, you should see something similar to this:

```
user@cf162-01:~\$ gdb test
GNU gdb (Ubuntu 7.10-1ubuntu2) 7.10
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-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 test...done.
```

Now gdb is up and running on our program test.c. If the copyright, license, and manual information is too abrasive, it is possible to simplify the start message with the -q argument:

```
user@cf162-01:~\$ gdb -q test
```

It is also possible to use GDB in the TUI mode. TUI stands for **T**ext **U**ser **I**nterface. It can display your code, breakpoints, and the current line quite nicely. TUI is launched by running gdb with the -tui argument:

```
user@cf162-01:~\$ gdb -tui test
```

The “run” (r) command is used to begin the program as you would from the terminal:

```
(gdb) r
```

To run the program with command line arguments, include them after “run” (r) just as we do in the terminal. Running the code as it is now is essentially the same as running it from the command line normally because it will run to termination. Luckily, GDB has tools that will allow us to pause and restart the code throughout the program run. This is accomplished with breakpoints which pause the code at key segments.

1.2 Making Break Points

Breakpoints play an essential role as we debug our code with GDB. They allow us to pause the code’s execution at functions, lines of code, and even memory addresses. To insert a break point into your code at a function or code line, use the command “breakpoint” (b):

```
(gdb) b main
Breakpoint 1 at 0x4005e4: file test.c, line 28.
```

This will cause our code to pause when it reaches the main() function. It can also be useful to set a break point at a particular line number of the code. In many cases we care more about getting to a specific point in the code rather than how we got there. GDB allows us to set a break point on a line of code per file. Let’s set two breakpoints on lines 29 and 32:

```
(gdb) b 29
Breakpoint 2 at 0x4005ec: file test.c, line 29.
(gdb) b 32
Breakpoint 3 at 0x400602: file test.c, line 32.
```

Alternatively, if we have multiple files, we will need to specify the exact file when setting the break point. Note the added “:” between the file name and line number:

```
(gdb) b test.c:32
Note: breakpoint 3 also set at pc 0x400602.
Breakpoint 4 at 0x400602: file test.c, line 32.
```

You will notice that we have created two breakpoints at the same position which isn't super helpful. To get a listing of the breakpoints, we can use "info breakpoints" (i b):

```
(gdb) i b
Num Type Disp Enb Address What
1 breakpoint keep y 0x00000000004005e4 in main at test.c:28
2 breakpoint keep y 0x00000000004005ec in main at test.c:29
3 breakpoint keep y 0x0000000000400602 in main at test.c:32
4 breakpoint keep y 0x0000000000400602 in main at test.c:32
```

This shows us that break points 3 and 4 are the same, and we can delete break point 4 using the "delete" (d) command:

```
(gdb) d 4
(gdb) i b
Num Type Disp Enb Address What
1 breakpoint keep y 0x00000000004005e4 in main at test.c:28
2 breakpoint keep y 0x00000000004005ec in main at test.c:29
3 breakpoint keep y 0x0000000000400602 in main at test.c:32
```

Note that "d 4" did not print anything, but we can verify that it worked with "i b" again.

There are additional ways to set breakpoints such as conditional variables, but that is out of the scope of this workshop. You can find information about advanced break point commands in the GDB Documentation (https://sourceware.org/gdb/current/onlinedocs/gdb/Breakpoints.html).

1.3 Traversing Code

Now that we have set a few break points, let's "run" (r) the program and see how to use the break points to navigate our code:

```
(gdb) r
Starting program: /home/user/test
Breakpoint 1, main () at test.c:28
28 char *a_word = "apricot";
```

It looks like we've hit our first break point on line 28. We use the command "next" (n) to move on and execute the next line of code:

```
(gdb) n
Breakpoint 2, main () at test.c:29
29 b();
```

This brings us to our second break point as well as a function call. At this point, we can either step into the function or step over the function Since it is not important for us to see exactly what happens in function b(), let's step over it with "next" (n):

```
(gdb) n
B is for Banana.
30 a(a_word);
```

The command “next” (n) is the major way you will move through your code in GDB. Sometimes we want to follow a function call by stepping into it with the “step” (s) command:

```
(gdb) s
a (word_a=0x40070d "apricot") at test.c:9
9 if (word_a == NULL || word_a[0] != ‘A’)
```

Now we’re inside of the a() function. We can continue to use the “next” (n) command to move through the function:

```
(gdb) n
10 printf("A is for Apple.\n");
```

While within a function, it is possible to allow the function to run automatically and for us to return to the caller. This can be completed with the “finish” command:

```
(gdb) finish
Run till exit from #0 a (word_a=0x40070d "apricot") at test.c:10
A is for Apple.
main () at test.c:31
31 c();
```

While the “next” (n) command takes us to the next line, we can also “continue” (c) to run through the code without stopping at each line:

```
(gdb) c
Continuing.
C is for Cantaloupe.
Breakpoint 3, main () at test.c:32
32 d();
```

It looks like we’ve hit our third and final break point. Let’s “continue” (c) again:

```
(gdb) c
Continuing.
D is for Dragonfruit.
Program received signal SIGSEGV, Segmentation fault.
0x000000000040060b in main () at test.c:34
34 a_word[0] = ’A’;
```

When we “continue” (c) to run this time, the program hits a segmentation fault, and a ha, we’ve found where our program crashes! It looks like we’ve erroneously tried to write to read only memory!

Let's fix our error and recompile the code. We can do this by replacing *a_word[0] = 'A'* with *a_word = "Apricot"*. Once we've recompiled, we can restart the program which will detect the file has changed and reload it for us. This can be done the same way we started the program initially, with the "run" (r) command:

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
'/home/user/test' has changed; re-reading symbols.
Starting program: /home/user/test
Breakpoint 1, main () at test.c:28
28 char *a_word = "apricot";
```

Although our previous example is slightly contrived, we hope this simple run exemplifies the power contained within GDB. There are many ways to examine variables of all types, pause code during run time, and step into specific functions of interest. Breakpoints allow us to look at the values of variables at any point in the code which will be even more useful as we encounter more complex programs and bugs. Let's take a look at some methods of determining how to solve the bugs we will inevitably encounter in the future.

Chapter 2

Examining Variables and Code

Now that we know how to run our program and set breakpoints, we can explore several methods to better understand what our code is doing. We will by compiling and loading GDB with the second sample code (test2.c) found in Chapter 3:

```
user@cf162-01:~$ gcc -g -o test2 test2.c
user@cf162-01:~$ gdb -q test2
Reading symbols from test2...done.
```

Begin by setting a breakpoint on main() and then "run" (r) the program.

```
(gdb) b main
Breakpoint 1 at 0x400650: file test2.c, line 16.
(gdb) r
Starting program: /home/user/test2
Breakpoint 1, main () at test2.c:16
16 int main(void) {
```

2.1 Display

First, we want to watch a variable change as we walk through the code. This can be done with the "display" command. Let's display variable word1 and step through a few of times:

```
(gdb) display word1
1: word1 = 0x4004e0 <_start> "1\355I\211\321^H\211\342H\203\344\360PTI \a@"
(gdb) n
17 char *word1 = "Hello";
1: word1 = 0x4004e0 <_start> "1\355I\211\321^H\211\342H\203\344\360PTI \a@"
(gdb) n
18 char word2[5] = {'H', 'e', 'l', 'l', 'o'};
1: word1 = 0x400783 "Hello"
```

Each time we step to the next line of code, we can see what the variable *word1* is equal to. Note that since this is the first variable we have displayed, it is prefaced with a "1:".

If we no longer want to display a variable, we can use the “undisplay” command as well:

```
(gdb) undisplay 1
```

The “1” comes from the beginning of the line showing *word1*. If you have multiple things displayed simultaneously, there will be multiple numbers shown and can all be turned off independently. Next, let’s “continue” (c) running the code and see what happens:

```
(gdb) c
Continuing.
Good is 5 characters long - "Hello"
Bad is 6 characters long - "Hello"
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400642 in words (good=0x400783 "Hello", bad=0x7fffffffe0a0
    "Hello\177") at test2.c:13
13 good[1] = 0;
```

Oh no! We hit a segfault! This says line 13 has caused it, but what else is in the code around there?

2.2 List

To determine what in the code has caused the segfault, we will use the “list” (l) command to see what is near line 13:

```
(gdb) l
8 printf("Good is %zu characters long - \"%s\"\n", length, good);
9
10 length = strlen(bad);
11 printf("Bad is %zu characters long - \"%s\"\n", length, bad);
12
13 good[1] = 0;
14 }
15
16 int main(void) {
17 char *word1 = "Hello";
```

2.3 Backtrace and Frame

Next, let's use the "backtrace" (bt) command to get a better understanding of our code:

```
(gdb) bt
#0 0x0000000000400642 in words (good=0x400783 "Hello", bad=0x7fffffffe0a0
    "Hello\177") at test2.c:13
#1 0x000000000040068e in main () at test2.c:20
```

Backtrace tells us that there are two frames on the call stack. We can use the "frame" (f) command to see what the state of either frame is:

```
(gdb) f 1
#1 0x000000000040068e in main () at test2.c:20
20 words(word1, word2);
```

2.4 Info Locals

While traversing a function, it can also be useful to see the variables present and their associated values. The "info local" command will do this for us:

```
(gdb) info local
word1 = 0x400783 "Hello"
word2 = "Hello"
```

Now let's go back to the frame that had the segfault and inspect around there some more.

```
(gdb) f 0
#0 0x0000000000400642 in words (good=0x400783 "Hello", bad=0x7fffffffe0a0
    "Hello\177") at test2.c:13
13 good[1] = 0;
```

2.5 Print

It may be helpful to "print" (p) the variables and see what values they hold:

```
(gdb) p good
$1 = 0x400783 "Hello"
(gdb) p bad
$2 = 0x7fffffffe0a0 "Hello\177"
```

We've got two problems here. Let's look at the addresses shown by print. The first is much lower than the second, but we declared them both in the same function! The problem here is that *good* lives in the read-only section of the binary and *bad* was allocated on the stack. Our second problem is with *bad*. It says the length is 6 in the printf call and print displays a weird "\177" at the end of it.

2.6 Examine

Now that we have determined where the issue with our code originates from, we can examine *bad* further with the “examine” (x) command:

```
(gdb) x/6bc bad
0x7fffffffe0a0: 72 'H' 101 'e' 108 'l' 108 'l' 111 'o' 127 '\177'
```

You might be wondering about the /6bc we typed after the x. This tells GDB to display 6 **b**ytes and to print them as **c**haracters. There are many more combinations of format code that can be used here. For more information, take a look at the table at `http://sourceware.org/gdb/onlinedocs/gdb/Output-Formats.html`. Examine shows us that we forgot to null terminate *bad* leading to our problem.

2.7 Disassemble

Sometimes, it can be useful to see what's happening at the machine level. We accomplish this by looking at the executable with the "disassemble" (disas) command. Let's restart the program and disassemble part of the main function:

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/test2
Breakpoint 1, main () at test2.c:16
16 int main(void) {
(gdb) disas
Dump of assembler code for function main:
   0x0000000000400648 <+0>: push %rbp
   0x0000000000400649 <+1>: mov %rsp,%rbp
   0x000000000040064c <+4>: sub $0x20,%rsp
=> 0x0000000000400650 <+8>: mov %fs:0x28,%rax
   0x0000000000400659 <+17>: mov %rax,-0x8(%rbp)
   0x000000000040065d <+21>: xor %eax,%eax
   0x000000000040065f <+23>: movq $0x400783,-0x18(%rbp)
   0x0000000000400667 <+31>: movb $0x48,-0x10(%rbp)
   0x000000000040066b <+35>: movb $0x65,-0xf(%rbp)
   0x000000000040066f <+39>: movb $0x6c,-0xe(%rbp)
   0x0000000000400673 <+43>: movb $0x6c,-0xd(%rbp)
   0x0000000000400677 <+47>: movb $0x6f,-0xc(%rbp)
   0x000000000040067b <+51>: lea -0x10(%rbp),%rdx
   0x000000000040067f <+55>: mov -0x18(%rbp),%rax
   0x0000000000400683 <+59>: mov %rdx,%rsi
   0x0000000000400686 <+62>: mov %rax,%rdi
   0x0000000000400689 <+65>: callq 0x4005d6 <words>
   0x000000000040068e <+70>: mov $0x0,%eax
   0x0000000000400693 <+75>: mov -0x8(%rbp),%rcx
   0x0000000000400697 <+79>: xor %fs:0x28,%rcx
   0x00000000004006a0 <+88>: je 0x4006a7 <main+95>
   0x00000000004006a2 <+90>: callq 0x4004a0 <__stack_chk_fail@plt>
   0x00000000004006a7 <+95>: leaveq
   0x00000000004006a8 <+96>: retq
End of assembler dump.
```

The "=>" arrow points to the current instruction that will be executed next. Keep in mind that there may be multiple instructions per line of C code. We can step through individual instructions by using the "stepi" (si). This will step one instruction at a time. It can be differentiated from step by the fact that it steps one assembly instruction and not one line of C code.

Sometimes you want to view the contents of registers just as if they were locals. This can be accomplished with the "info registers" (i r) command.

```
(gdb) i r
rax 0x400648 4195912
rbx 0x0 0
rcx 0x0 0
rdx 0x7fffffffe838 140737488349240
rsi 0x7fffffffe828 140737488349224
rdi 0x1 1
rbp 0x7fffffffe740 0x7fffffffe740
rsp 0x7fffffffe720 0x7fffffffe720
r8 0x400720 4196128
r9 0x7ffff7de7ab0 140737351940784
r10 0x846 2118
r11 0x7ffff7a2d740 140737348032320
r12 0x4004e0 4195552
r13 0x7fffffffe820 140737488349216
r14 0x0 0
r15 0x0 0
rip 0x400650 0x400650 <main+8>
eflags 0x202 [ IF ]
cs 0x33 51
ss 0x2b 43
ds 0x0 0
es 0x0 0
fs 0x0 0
gs 0x0 0
```

While there are many more things that GDB can do and help you with we don't have time to talk about them all. In the Further Reading section you will find links to many helpful GDB pages.

Chapter 3

Sample Code

This workshop uses the following sample codes. To access them on GitLab, you will need to enter your WWU CS username and password. The code is also printed in plaintext on the following pages for reference.
Chapter 1 - `https://gitlab.cs.wwu.edu/wwucs_mentors/wwucs_workshops/raw/master/gdb/test.c`
Chapter 2 - `https://gitlab.cs.wwu.edu/wwucs_mentors/wwucs_workshops/raw/master/gdb/test2.c`

test.c

```c
#include <stdio.h>

void a(char *word_a);
void b(void);
void c(void);
void d(void);

void a(char *word_a) {
  if (word_a == NULL || word_a[0] != 'A')
    printf("A is for Apple.\n");
  else
    printf("A is for %s\n", word_a);
}

void b(void) {
  printf("B is for Banana.\n");
}

void c(void) {
  printf("C is for Cantaloupe.\n");
}

void d(void) {
  printf("D is for Dragonfruit.\n");
}

int main(void) {
  char *a_word = "apricot";
  b();
  a(a_word);
  c();
  d();
  /* Capitalize the first letter */
  a_word[0] = 'A';
  /* The statement above will segfault */
  a(a_word);
  return 0;
}
```

test2.c

```c
#include <stdio.h>
#include <string.h>

void words(char *good, char *bad) {
  size_t length;

  length = strlen(good);
  printf("Good is %zu characters long - \"%s\"\n", length, good);

  length = strlen(bad);
  printf("Bad is %zu characters long - \"%s\"\n", length, bad);

  good[1] = 0;
}

int main(void) {
  char *word1 = "Hello";
  char word2[5] = {'H', 'e', 'l', 'l', 'o'};

  words(word1, word2);
  return 0;
}
```

Chapter 4

Further Reading

- https://www.gnu.org/software/gdb/documentation/
- http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf
- http://www.yolinux.com/TUTORIALS/GDB-Commands.html