Notes and Tutorial on GDB

Jack Rosenthal

CSM Linux Users Group

Note

This isn't a complete reference, but an unprofessional handy source. You might find out a lot more by reading the manual.

Contents

1 Introduction

The GNU Debugger (GDB) is an awesome tool for debugging your programs written in C, C++, FORTRAN (please don't), and plenty of other compiled languages. You can use it by running `gdb` from the command line.

1.1 Compiling your programs for GDB

GDB can make use of special symbols in your program to help you debug. Add the `-ggdb3` flag to include these symbols. Example below:

```
$ gcc -ggdb3 -std=c11 -o program.debug program.c
```

This compiler flag works with `gcc`, `clang`, or any of their related compilers (`g++`, etc.). Generally, you will only want to use this flag for debugging, as it not only embeds the source of your program in the binary, but also increases the size of the binary significantly. The rest of this document assumes you have compiled with this flag when debugging.

Many people add a `debug` target to their `Makefile`, this allows them to compile their program for debugging and start the debugger simply typing `make debug`. The following listing contains a complete `Makefile` example, it may be overkill for small projects.

Listing 1: A complete `Makefile` example

```
CXX = clang
SRCFILES = $(wildcard src/*.c)
OBJFILES = $(patsubst src/%.c, bin/%.o, $(SRCFILES))
OBJDEBUG = $(patsubst src/%.c, bin/%.debug.o, $(SRCFILES))
OUTFILE = myprogram
CXXFLAGS = -Wall -std=c11
LDFLAGS = -lreadline

all: $(OUTFILE)

$(OUTFILE): $(OBJFILES)
        $(CXX) $(LDFLAGS) $(OBJFILES) -o $(OUTFILE)

bin/%.o: src/%.c
        $(CXX) $(CXXFLAGS) -o $@ -c $<

# 'make debug' depends on myprogram.debug. Make will check if this file
# exists and is up to date, and if so, call the debugger.
debug: $(OUTFILE).debug
        gdb -tui $(OUTFILE).debug

# 'make myprogram.debug' links the binary we are looking for.
$(OUTFILE).debug: $(OBJDEBUG)
        $(CXX) $(LDFLAGS) $(OBJDEBUG) -o $(OUTFILE).debug

# This makes each individual object with debugging symbols
bin/%.debug.o: src/%.c
        $(CXX) $(CXXFLAGS) -ggdb3 -o $@ -c $<
```

This `Makefile` compliles a multifile project with all of the source located at `src/*.c`.

1.2 A basic example

A coworker has handed you a simple C program they wrote, but is segfaulting. They are fairly new to C programming[1] and would like your help debugging. The program they wrote follows:

Listing 2: Your coworker's program (`prog1.c`)

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

/**
 * Allocate memory and read a string from standard input
 */
char *
read_str_buf (size_t buf_sz)
{
    char    *buf;
    buf = malloc(buf_sz);
    buf = fgets(buf, buf_sz - 1, stdin);
    return buf;
}

int
main (argc, argv)
    int     argc;
    char    **argv;
{
    char    *str;
    for (;;) {
        printf("What would you like me to print? ");
        str = read_str_buf(1<<31);
        printf("%s\n", str);
    }
    free(str);
    return 0;
}
```

Can you spot the bug? Not easy without a debugger. Let's throw it in `gdb`! Start off by compiling with debugging symbols and starting `gdb` on the program.

```
$ gcc -ggdb3 -o prog1.debug prog1.c
$ gdb prog1.debug
```

[1] Your coworker also used the 1st edition K&R C book to learn

When `gdb` first starts, you will see this:

```
GNU gdb (GDB) 7.10.1
Copyright (C) 2015 Free Software Foundation, Inc.
...
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from prog1.debug...done.
(gdb)
```

`gdb` has near a gazillion commands, but we are going to want to use either `start` or `run` here. `start` will create a breakpoint at the start of `main` before running, whereas `run` will run right away. If we wanted the program to read a file on standard input, we could use the syntax `run < myfile` to read it. Go ahead and type `run`, and give it the input "`Hello, World!`":

```
(gdb) run
Starting program: prog1.debug
What would you like me to print? Hello, World!

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7aa0eb0 in __GI__IO_getline_info () from /usr/lib/libc.so.6
```

Yikes! What caused that? We can generate a backtrace to see where this happened. To do this, type `backtrace` (or, if you are lazy, `bt` works too).

```
(gdb) bt
#0 0x00007ffff7aa0eb0 in __GI__IO_getline_info () from /usr/lib/libc.so.6
#1 0x00007ffff7a9fe1d in fgets () from /usr/lib/libc.so.6
#2 0x0000000000400650 in read_str_buf (buf_sz=...) at prog1.c:12
#3 0x0000000000400680 in main (argc=1, argv=0x7fffffffe7e8) at prog1.c:24
```

Well, obviously it was at that `fgets` on line 12. Let's `kill` the program and restart it. We'll `break` at the start of `read_str_buf` to inspect what may be happening there.

```
(gdb) break read_str_buf
Breakpoint 1 at 0x400622: file prog1.c, line 11.
(gdb) run
Starting program: prog1.debug

Breakpoint 1, read_str_buf (buf_sz=...) at prog1.c:11
11          buf = malloc(buf_sz);
```

Next, let's print some variables and step through some lines. We are going to use a few commands:

`print` *varname*	Print the value of *varname*, can be abbreviated to `p`
`next`	Go to the next line, can be abbreviated to `n`

```
(gdb) print buf_sz
$1 = 18446744071562067968
(gdb) next
12          fgets(buf, buf_sz, stdin);
(gdb) print buf
$2 = 0x0
(gdb) next
What would you like me to print? Hello, World?

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7aa0eb0 in __GI__IO_getline_info () from /usr/lib/libc.so.6
```

Hopefully you said "Hey that's it, `malloc` returned `NULL`" when you printed `buf`. You can go ahead and type `quit` now, we will come back to this example later.

Brownie points are available if you can name why:

1. Even if the program worked, and allocated a reasonable amount of memory (say 1024), it would still leak memory.
2. The prompt line did not print until we reached the `fgets`
3. `buf_sz` was 18446744071562067968, not 2147483648
4. If line 12 was changed to read:

   `buf = fgets(buf, buf_sz, stdin);`

   the program would not segfault until line 25.

1.3 The Terminal User Interface

Yes. `gdb` has a nice terminal user interface that you can access by starting `gdb` with the `-tui` flag. This allows you to view each line in the source code you are stepping through in a nice box above the debugger. Sometimes the interface will get messed up and you will need to redraw, simply type Ctrl+L to redraw it.

2 Basic Features

2.1 Stepping through your code

For stepping through your code, you will want to use the `step`, `next`, and `continue` (abbreviated to `s`, `n`, and `c`) commands.

- The `step` command will execute whatever is on the current line *stepping into function calls* if there are any.

- The `next` command will execute whatever is on the current line *continuing through function calls* if there are any.
- The `continue` command will keep running your code until it reaches a breakpoint.

Using the program from earlier, here is an example. You may find it helpful to use the TUI, so you can see which line you are at visually.

```
(gdb) start
Temporary breakpoint 1 at 0x400669: file prog1.c, line 23.
Starting program: prog1.debug

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffe7e8) at prog1.c:23
23              printf("What would you like me to print? ");
(gdb) next
24              str = read_str_buf(1<<31);
(gdb) step
read_str_buf (buf_sz=18446744071562067968) at prog1.c:11
11          buf = malloc(buf_sz);
(gdb) next
12          buf = fgets(buf, buf_sz - 1, stdin);
(gdb) next
What would you like me to print? Hello, World?

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7aa0eb0 in __GI__IO_getline_info () from /usr/lib/libc.so.6
```

Notice how we used `step` to step into the `read_str_buf` function call. If we used `next`, it would silently run that function without taking us through it.

2.2 Working with Variables

`gdb` gives you full control of your variables at any time.

`print` *varname*	Print the value of *varname*, can be abbreviated to `p`
`set` *varname*`=`*value*	Set *varname* to *value*
`print` *varname*`=`*value*	Set *varname* to *value* and print it

value can be another variable.

To change frames, type `backtrace` to find the frame number to change to, then type `frame` *n* to change to frame *n*.

To list all variables in this frame, type `info locals`. To list the arguments to this frame, type `info args`. To get way too much information about this frame, type `info frame`.

Listing 3: `prog2.c`

```c
#include <stdio.h>
#include <stdlib.h>
extern char **environ;

/**
 * My own strlen implementation
 */
int my_strlen(const char *in) {
    const char *p = in;
    while (*p++);
    return p - in - 1;
}

/**
 * My own strdup implementation
 * I think there's some sort of bug in this... but I can't find it!
 */
char * my_strdup(const char *in) {
    char *new = malloc(my_strlen(in) + 1), *p = new;
    while (*in) {
        *p = *in;
        p++, in++;
    }
    return new;
}

int main(int argc, char **argv) {
    char **p = environ;
    while (*p) {
        char *str = my_strdup(*p);
        printf("%s\n", str);
        free(str);
        p++;
    }
    return 0;
}
```

2.3 Breakpoints

To set a breakpoint, use the incredibly powerful `break` command. The general syntax is:

`break [`⟨location⟩`] [if` ⟨condition⟩`]`

where ⟨location⟩ can be:

A line number (current file assumed)	`lineno`
A file name and line number	`file.c:lineno`
A function name	`my_function`
A function name with namespace (for C++)	`std::my_function`
A function with types (for overloading in C++)	`my_function(int)`
The address of the program counter	`*0xfedcba76`

and ⟨condition⟩ can be any valid conditional syntax in your language (eg. `i == 32`). If the location is not specified, the current program counter is assumed.

For our example, we are going to use the `prog2.c` program, which reimplements `strdup`, but with a bug. The `main` function prints all of the user's environment variables after they've been through `my_strdup` so you can see the bug (compare to the output of `env`).

Load up this program in `gdb` and issue the following commands. Keep pressing `c` whenever you reach a breakpoint, and notice how breakpoint 2 will only trigger when the first letter of the environment variable is H. Neato!

```
(gdb) break my_strlen
(gdb) break my_strdup if *in == 'H'
(gdb) run
```

2.4 Watchpoints

You may want to break the program whenever a variable is changed or read. This is called setting a watchpoint. The syntax is:

`watch` ***x***	break whenever ***x*** is changed.
`rwatch` ***x***	break whenever ***x*** is read.
`awatch` ***x***	break upon read or write

2.5 Disabling and Enabling {Break,Watch}points

If you are tired of breaking, you may want to disable a specific breakpoint. To list breakpoints and their associated number ***n***, type `info breakpoints`. Then, type `disable` ***n*** or `enable` ***n***.

3 Advanced Magical Wizardry

3.1 Calling Functions

`gdb` allows you to call one of your functions, or any function linked to your program. Use the `call` command. Here is an example:

```
(gdb) start
Temporary breakpoint 1 at 0x4006b7: file prog2.c, line 28.
Starting program: prog2.debug

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffe7c8) at prog2.c:28
28          char **p = environ;
(gdb) call my_strlen("Hello, World!")
$1 = 13
```

3.2 Examining Memory

`gdb` has an amazing command, `x`, but few people actually remember how to use it without typing `help x`. Here is its syntax:

`x/`⟨amount⟩⟨format⟩⟨size⟩ ⟨variable or address⟩

⟨amount⟩ is the amount of objects of the specified size to print. It will default to 1 if you don't specify it.

⟨format⟩ is a single character that specifies how it should look when printed. Here are the different options:

- `o` octal
- `x` hexadecimal
- `d` decimal
- `u` unsigned decimal
- `t` binary
- `f` float
- `a` address
- `i` instruction
- `c` char
- `s` string
- `z` zero padded hexadecimal

⟨size⟩ specifies the size of the data to be printed. It is also a single character. Here are the different options:

- `b` byte (8 bits)
- `h` half word (16 bits)
- `w` word (32 bits)
- `g` giant word (64 bits)

That was awfully complex. For an example, let's examine `prog3.c`.

Listing 4: `prog3.c`

```c
#define _POSIX_C_SOURCE 200809L
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char *str = strdup("Hello, World!\n");
    int x = 42;
    int *y = &x;
    double pi = 3.1415926535897932384;
    double *pi_ptr = &pi;
    if (*str && x && *y && *(int *)pi_ptr)
        printf("Goodbye, world!\n");
    return 0;
}
```

Here are some examples:

```
(gdb) start
Temporary breakpoint 1 at 0x400555: file prog3.c, line 6.
Starting program: prog3.debug

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffe7c8) at prog3.c:6
6           char *str = strdup("Hello, World!\n");
(gdb) n
7           int x = 42;
(gdb) x/s str
0x601010:       "Hello, World!\n"
(gdb) x/c str
0x601010:       72 'H'
(gdb) x/6c str
0x601010:       72 'H'  101 'e' 108 'l' 108 'l' 111 'o' 44 ','
(gdb) n
8           int *y = &x;
(gdb) n
9           double pi = 3.1415926535897932384;
(gdb) x/tw y
0x7fffffffe6c4: 00000000000000000000000000101010
(gdb) n
10          double *pi_ptr = &pi;
(gdb) n
11          if (*str && x && *y && *(int *)pi_ptr)
(gdb) x/tg pi_ptr
0x7fffffffe6b8: 0100000000001001001000011111101101010100010001000010110100011000
```

3.3 When things get so bad to the point where you are dealing with assembly code

You may want to `disassemble` part of the program:

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000400546 <+0>:     push   %rbp
   0x0000000000400547 <+1>:     mov    %rsp,%rbp
   0x000000000040054a <+4>:     sub    $0x40,%rsp
   0x000000000040054e <+8>:     mov    %edi,-0x34(%rbp)
...
```

Maybe examine the registers:

```
(gdb) info registers
rax            0x601010         6295568
rbx            0x0              0
rcx            0xa21646c726f57  2851464966991703
rdx            0x0              0
rsi            0x400657         4195927
...
```

You can also use `nexti` and `stepi` to step instructions (rather than lines).

You my friend, have achieved GDB wizard status.