Chapter 3

Debugging with GDB

Motivation

You thought you were getting a bargain when you bought that pocket watch from that scary old lady at the flea market in the French Quarter. Little did you realize that pocket watch did more than just tell time.[1] In fact, holding the button on the side *slows down* time.

At first it's a novelty. You prank your roommate. Maybe take some extra time on a test. Steal a quick bagel for breakfast. Rob a bank...[2]

It was all fun up to this point. But now you're here: standing in front of an array of red-hot lasers, trapped in the vault of the Big Bank of New York, wondering where it all went wrong, and wearing a fancy turtle neck. At least you look good in a turtle neck. Not everyone can pull that off.

Desperate for a way out, you start mashing and twisting the other buttons and dials on your magic pocket watch.

BWEEEEEEEEEEEEEEEE![3]

With a deafening *bwee*, you're hurled back in time. Once again, you're crouched in front of the control panel for the Big Bank's vault. You stare, bewildered, at the room around you.

After you gather yourself, you analyze the nest of wires pouring from the control panel and wonder what you did to set off the vault's laser defense system. You slow time and begin cutting the wires with the procedure you used last time. Red wire, other red wire, otherer red wire. You look behind you: no lasers – everything's good.

Blue wire.

The array of lasers begins slowly cutting across the doorway. You're trapped again. With your last few seconds of freedom, you look in the panel and find *another blue wire*. You cut the wrong blue wire!

[1] Actually it doesn't tell time at all. The hands didn't move when you bought it, and they still don't. It sure looks cool, though.
[2] That escalated pretty quickly.
[3] *BWEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE*

You reach into your pocket to grab your watch and reset time. Instead, you pull out that pastrami sandwich from last lab. Also, you've turned into a raccoon somehow.

You wake up.[4]

In real life, you may not be able to slow down or reset time to analyze disasters in detail.[5] You can, however, slow down disasters as they happen in your programs.

Using a **debugger**, you can slow down the execution of your programs to help you figure out why it's not working. A debugger can't automatically tell you what's broken, but it can...

- step through your code line by line
- show you the state of variables
- show you the contents of memory

...so that you can figure out what's wrong on your own.

In your bank robber dream, the watch allowed you to slow down time to see what set off the laser defense system. In real life, a debugger allows you to slow down the execution of your program, so that you can see where and when it breaks. It's still up to you to figure out why it's breaking and how to fix your bugs, but a debugger can definitely give you some clues to make it easier to find them.

We'll be using the GNU Debugger (gdb) to debug a couple of C++ programs. gdb, like g++, is open source software. There are GUI frontends available for gdb, but in this chapter, we'll be using the command line interface.

Takeaways

- Learn general debugging practices
- Step through the execution of a compiled C++ program
- Inspect the contents of program variables

Walkthrough

Fatherly Debugging Advice

It's a sunny afternoon, so I take off work early and come by to pick you up from school. You run up and hop in the front seat of the car. "Hey kiddo, how was school? Do you want to get some ice cream?"

We drive over to the ice cream shoppe and I pull the ol' steering-with-my-knees trick that always makes you scream. "Dad! Stop it, you'll hit someone!" I laugh and mock-begrudgingly put my hands back on the wheel.

We sit down in a booth to enjoy a couple of sundaes. "How was your day? Are the other kids treating you well?" I ask.

[4]It was a dream that whole time. What a slap in the face!
[5]In real life, hopefully you're not a bank robber either.

“School was fine. I did great on my English paper! But…” You furrow your brow.

“What’s on your mind?”

“Well, some of my friends have been programming. And sometimes their programs don’t work, and they have to fix them. And I guess I’m worried that when my program doesn’t work I won’t be able to fix it!”

“My child,” I begin.

I put a hand on your shoulder the way only a father can.

“My dearest,”

I put another hand on your shoulder.

“It seems like only yesterday you were but a wee babe. How you did cry and scream and do other things that babies do! I lost more sleep over you than I did working 26 hour shifts at the smoke alarm testing factory.”

I put another hand on your shoulder.

“But there was one thing that always calmed you down. The instant I’d sit you next to a computer you’d smile and coo! Many an evening I rocked you to sleep cradled between the screen and keyboard of a Thinkpad 701C. That’s when I knew you’d grow up to be a great programmer.”

I put another hand on your shoulder. You are visibly embarrassed at this point.

“And now look at you! Practically all grown up! Your mother and I are so proud of you. Everyone has trouble with programs from time to time.”

I put another hand on your shoulder.

“Here’s what I do when I have to fix a program:”

1. What happened? What should have happened?

You should have an idea of how your program ought to work in your mind while programming. Maybe you’ve got a homework assignment writeup to go off of, or a design document that you wrote, or just an idea in your head that you’re implementing.

Regardless of where that idea comes from, take a second to identify the wrong behaviour you’re seeing and to figure out what your program should have done instead. If you can, get a real piece of paper and actually draw out a picture of what your program is doing; for example, if you’re making a binary tree, draw the tree, or if you’re communicating to a server, draw/write out the protocol your server and client should be following.

2. What the heck did you do to break it?!

Was your program working earlier? If so, think through the changes you just made. (You may find consulting `git diff` to be helpful!) How might they have caused the bug?

(If your program didn't work right in the first place, you'll have to try some other debugging technique. Sorry.)

3. How is the program getting the wrong result?

Start from where you saw the bug and work your way back through your program. What execution path did it follow? What values do your variables have? How do those values get set?

Make generous use of print (`cout`) statements! Print out the contents of your variables and print out messages that tell you what code is executing. Don't take anything for granted! Use those print statements to check your assumptions about how your code works.

4. Talk it out.

On the next page you will find Bertha.[6] She's very curious about people and would love to hear about your program. So, prop that page up next to your computer and explain exactly what is wrong with your program, just like you would explain it to a friend. Why do you think it's broken? What fixes did you try?

(This may seem foolish, but this technique once revealed to me that I had been debugging code for an hour under the misapprehension that 9 was a prime number.[7] You'd be amazed at the things you don't realize until you think about how to explain something to someone else.)

5. When all else fails, try again later.

If you've got a bug that you just can't seem to figure out, take a break. If there are other obvious bugs, fixing those may reveal the root cause of your problem bug. Otherwise, take a walk, or even better, get a good night's sleep and come back to it in the morning.

These debugging tips work for any program (and plenty of not-computer-related problems to boot). However, a debugger tool can make life easier for you, especially when you are tracing the execution of your program and inspecting how it behaves.

Compilin' for Debuggin'

Compiled programs don't contain all the information that their source code does. All of the C++ code that you write gets translated into machine-executable code. As far as your CPU is concerned, there's really no need for function names, curly braces, or comments. So, if you try to debug a program that's compiled in the usual manner, you'll have to wade through a lot of hex numbers and assembly code.

As a human person, you'd rather be able to look at your actual source code in the debugger. Fortunately, there is a way to ask `g++` to to keep the details of our source code when we compile it. Whenever you compile your code, simply add the `-g` flag to your `g++ command`. For example:

[6] Well, a picture of her. The publisher wouldn't let us include a live chicken tucked between the pages of the book.
[7] In my defense, odd numbers between 2 and 10 have a 75% chance of being prime, so I was statistically correct!

Figure 3.1: Bertha!

```
$ g++ -g main.cpp
```

If you forget the -g, gdb will have a lot less information to work with. As a result, debugging will be much less fun.

Starting GNU Debugger

Alrighty, friend. Let's get our hands dirty.

Pop open an editor and drop in the following C++ program. Save it as `segfault.cpp`. Make sure you keep all the whitespace the same — we'll be referring to the line numbers when we use gdb.

```
1  #include<iostream>
2  using namespace std;
3
4  struct my_array {
5    int* elements;
6    int size;
7  };
8
9  void pretty_print(const my_array& array) {
10   int i = 0;
11   cout << "[";
12
13   for(; i < array.size - 1; i++) {
14     cout << array.elements[i] << ", ";
15   }
16   cout << array.elements[array.size] << "]";
17 }
18
19 int main() {
20   my_array stuff;
21   stuff.size = 5;
22   stuff.elements = 0;
23
24   pretty_print(stuff);
25
26   return 0;
27 }
```

Now let's compile it and run it in gdb.

```
$ g++ -g -o segfault segfault.cpp
$ gdb segfault
GNU gdb ...
```

```
Reading symbols from segfault...done.
(gdb)
```

gdb will show you a *bunch* of license information and other junk before it give you its command prompt. You can ignore that stuff.

You're now in gdb! Here, you can issue commands to gdb, and it will do your bidding. You can step through your program, run parts of it at full speed, and check the values of different variables.

GNU Debugger Commands

Running programs

Assuming you followed the directions in the previous section, you should be sitting at the (gdb) prompt ready to run your first command.

Try issuing the `run` command. Just type `run` and press enter.

The `run` command will start your program and execute it at full speed until it reaches a stopping condition. Stopping conditions include:

1. reaching the end of a program.
2. reaching a breakpoint.
3. encountering a fatal error (like a segmentation fault).

Once gdb stops, it will tell you which of these stopping conditions it hit. For example, if your code segfaults, it'll print something like

```
Program received signal SIGSEGV, Segmentation fault.
0x555555554923 in pretty_print (array=...) at segfault.cpp:14
14    cout << array.elements[i] << ", ";
```

In order to debug your program, it has to be running. The `run` command is likely one of the first commands you will run whenever you debug a program.

If you need to pass any command line arguments to the program, you'll pass them to the run command. For example, if we wanted to debug g++…[8]

```
$ gdb g++
(gdb) run funcs.cpp main.cpp
```

…would be similar to running `g++ funcs.cpp main.cpp` outside of gdb.

Where are we?

Once gdb stops executing your code, you can use the `backtrace` (or `bt` for short) command to ask gdb where you currently are.

[8]Don't actually do this, though. We're just demonstrating how you'd pass command line arguments.

```
(gdb) bt
#0  0x555555554923 in pretty_print (array=...) at segfault.cpp:14
#1  0x5555555549b3 in main () at segfault.cpp:24
```

The backtrace shows you the function stack: starting from `main()`, which functions were called to get to the line that's currently executing? Think of it like a family history: this line of code lives in this function, which was called by this other function, which was called by *another* function, and so on and so forth all the way back to Adam, err, `main()`.

You can also see the file name and line number where the function is defined, along with the values of any arguments passed to it.

`backtrace` is usually the first tool you should reach for when debugging a segfault. It'll tell you right where your program is, and it might even show you the null pointer (if you're passing it as an argument to the function).

Each entry in the function stack is called a **stack frame**. By default, `gdb` selects the frame at the top of the stack for inspecting. We can use the `info` command to inspect the selected frame's **arguments** (`info args`) and **local variables** (`info locals`):

```
(gdb) info args
array = @0x7fffffffe3d0: {elements = 0x0, size = 5}
(gdb) info locals
i = 0
```

(Here we can see the null pointer that's giving us a segfault!)

You can use `list` to ask `gdb` to show you some source code to give you context. Sometimes a line number isn't enough if you're too lazy to tab over to your text editor. It'll list the source code for the selected frame by default.

```
9   void pretty_print(const my_array& array) {
10    int i = 0;
11    cout << "[";
12
13    for(; i < array.size - 1; i++) {
14      cout << array.elements[i] << ", ";
15    }
16    cout << array.elements[array.size] << "]";
17  }
```

An interesting quirk about `list` is that it will continue to show more lines if you run it again. If it runs out of lines to show you, it becomes grumpy.

You can change the selected stack frame with the `up` and `down` commands. Notice how the stack frame at the top of the stack (in our example, the one for `pretty_print`) is numbered `0`, the next (`main`) is numbered `1`, and so on. `up` increments the number of the selected stack frame; `down` decrements it.

So, for example, to move to `main`'s stack frame, we'd do:

```
(gdb) up
#1  0x5555555549b3 in main () at segfault.cpp:24
```

```
24    pretty_print(stuff);
```

Now we can use `info locals` to inspect the variables declared in `main`:

```
(gdb) info locals
stuff = {elements = 0x0, size = 5}
```

In case you get lost, `frame` or `f` will tell you which frame you've currently selected.

Stopping at the right time

If you want your program to pause at a specific line, you can place a **breakpoint**. Let's use a more complex example to demonstrate breakpoints:

```cpp
#include<iostream>
using namespace std;

bool check_guess(const int guess, const bool new_answer) {
  static int answer;

  if(new_answer) {
    // Pick a number between 1 and 10
    answer = 1 + rand() % 10;
  }

  return guess == answer;
}

void play_game() {
  int guess;
  bool got_it = true;

  do {
    cout << "Enter your guess: ";
    cin >> guess;

    got_it = check_guess(guess, got_it);

    if(got_it) {
      cout << "You got it!" << endl;
    }
    else {
      cout << "NRRRRRR wrong! Try again!" << endl;
    }
  } while(!got_it);
}

int main() {
```

```
35    char again;
36
37    do {
38      play_game();
39
40      cout << "Want to play again? (y/n) ";
41      cin >> again;
42    } while(again == 'y' || again == 'Y');
43
44    return 0;
45  }
```

Let's say we want to step through the execution of `play_game`. *Before* typing `run`, set a breakpoint with the `break` command, *then* run your program:

```
(gdb) break 15
Breakpoint 1 at 0xaf6: file guess.cpp, line 15.
(gdb) run
Starting program: guess

Breakpoint 1, play_game () at guess.cpp:15
15  void play_game() {
```

Our program *is running*, but `gdb` has paused its execution at line 15.

You can set as many breakpoints as you like. The `info breakpoints` command will show you a list of all the breakpoints you've set.

If you find that you have too many breakpoints, or if they're getting in the way, you can delete them using the `delete` command. By itself, `delete` will delete all breakpoints. If you want to delete a specific breakpoint, you can refer to it by its ID number. For example:

```
(gdb) break 17
Breakpoint 2 at 0x555555554b05: file guess.cpp, line 17.
(gdb) break 19
Breakpoint 3 at 0x555555554ba0: file guess.cpp, line 19.
(gdb) info breakpoints
Num Type       Disp Enb Address            What
1   breakpoint keep y   0x555555554af6 at guess.cpp:15
2   breakpoint keep y   0x555555554b05 at guess.cpp:17
3   breakpoint keep y   0x555555554ba0 at guess.cpp:19
(gdb) delete 2
(gdb) info breakpoints
Num Type       Disp Enb Address            What
1   breakpoint keep y   0x555555554af6 at guess.cpp:15
3   breakpoint keep y   0x555555554ba0 at guess.cpp:19
```

Stepping through the code

Now that we've used a breakpoint to pause our code, we can step through the execution. There are a handful of commands that will help us do this:

- `continue`: Resume `run`ning the program at full speed. We'll only stop/pause execution if we reach a stopping condition as described for `run`.
- `step`: Runs *one* line of code, stepping *into* function calls. If you reach a function call, `step` will enter that function.
- `next`: Runs *one* line of code, stepping *over* function calls. If you reach a function call, `next` will run the entire function until it returns.
- `finish`: Runs code until the current function returns.

If we run `step` once, we'll see that it runs line 16 and stops:

```
(gdb) step
17    bool got_it = true;
```

Looking at contents of variables

`gdb` also allows you to look at what's stored in different variables with the `print` command. This can be a *very* handy alternative to placing `cout` statements all over the place.

Let's use `print` to look at the contents of `got_it`:

```
(gdb) p got_it
$1 = false
```

Well that's interesting, isn't it? `false` is not `true` at all! That's because at this point, `gdb` hasn't actually run line 17 yet. Let's step forward and check it again.

```
(gdb) step
20      cout << "Enter your guess: ";
(gdb) p got_it
$2 = true
```

That makes a lot more sense!

You can use `print` to print out any variable that's in scope. It even knows how to print out struct and class instances! (Try it out on the `segfault` program from earlier!) `print` will also happily print out the results of C++ expressions. You can dereference pointers, do arithmetic, and even call functions!

Let's use `print` to cheat at our guessing game. We want to see what the value of `answer` gets set to in `check_guess`. Let's advance the program's execution to that point!

First, notice that we're about to run line 20, which is going to call `operator<<` and probably a handful of other things in the standard library that we don't care about and really don't want to step through one line at a time. So, let's run `next` twice to execute lines 20 and 21:

```
20      cout << "Enter your guess: ";
(gdb) next
```

```
21      cin >> guess;
(gdb) next
Enter your guess: 5
23      got_it = check_guess(guess, got_it);
```

Don't get too `next`-happy though, because we sure do want to step into `check_guess`! `step` into `check_guess` and keep `step`ping until `answer` gets set, then print out the value:

```
23      got_it = check_guess(guess, got_it);
(gdb) step
check_guess (guess=5, new_answer=true) at guess.cpp:7
7     if(new_answer) {
(gdb) step
9       answer = 1 + rand() % 10;
(gdb) step
12    return guess == answer;
(gdb) print answer
$3 = 4
```

Aha! The secret we need to "guess" is 4! Now we can use `continue` to resume executing the program and win:

```
(gdb) continue
Continuing.
NRRRRRR wrong! Try again!
Enter your guess: 4
You got it!
Want to play again? (y/n)
```

We could use `print` to check a bunch of guesses without getting taunted by the mean program:

```
(gdb) print check_guess(7, false)
$4 = false
(gdb) print check_guess(4, false)
$5 = true
```

Setting the contents of variables

Sometimes you'll want to test out a theory — maybe you think you've found your bug, and you want to make a quick change to see if you know how to fix it. You can't change the code while `gdb` is running it, but you can poke stuff in memory to your heart's content.

You can change the values of variables on the fly with the `set` command!

Let's use this to cheat a different way. Play the game again (i.e., press `y`). `gdb` will stop us when we hit our breakpoint on line 15. The `advance` command works like a one-time breakpoint; use it to run the program until line 25:

```
(gdb) advance 25
Enter your guess: 3
```

```
play_game () at guess.cpp:25
25      if(got_it) {
```

Now, let's see if our guess was right and use `set` to make our guess "right" if it wasn't!

```
(gdb) p got_it
$6 = false
(gdb) set got_it = true
(gdb) continue
Continuing.
You got it!
Want to play again? (y/n)
```

Alright, that's enough cheating for one school day.

```
Want to play again? (y/n) n
[Inferior 1 (process 24164) exited normally]
(gdb) quit
```

There's a lot more that `gdb` can do — disassembling functions, poking around in memory, debugging code running on other machines — but the stuff in this chapter will cover all of your usual debugging needs. Happy debugging!

Questions

Name: `______________________________`

1. In your own words, what does `backtrace` do?

2. How would you set a breakpoint for line 17 of file `my_funcs.cpp`?

3. Consult `help list`. How would you list the source code of a function named `encabu-late_beziers`?

4. What is the difference between `break` and `advance`?

Quick Reference

Using gdb

- `gdb PROGRAM` launches the debugger for debugging `PROGRAM`.
  - **Note**: You will want to pass `g++` the `-g` flag when you compile!
- `run arg1 arg2` runs the command with command line arguments `arg1` and `arg2`.
- `backtrace` or `bt` shows the call stack.
- `up` and `down` change which stack frame is selected for `info` commands.

Setting breakpoints

- `break main.cpp:10` will stop execution whenever line 10 in `main.cpp` is reached.
- `advance main.cpp 15` will stop execution the next time line 15 in `main.cpp` is reached.
- `continue` resumes running as normal.
- `step` runs one more line of code, stepping **into** functions as necessary.
- `next` runs until execution is on the next line, stepping **over** functions.
- `finish` runs until the current function returns.
- `delete` removes all breakpoints.

Looking at variables

- `print VARIABLE` prints the contents of variable with name `VARIABLE`.
  - `p VARIABLE` also works.
  - `p` also works with expressions of just about any sort.
- `info locals` shows variables declared in the current function.
- `info args` shows arguments passed into the current function.
- `info registers` lists all CPU register values
- `p $REGNAME` prints the value of a CPU register

Miscellaneous

- Conditional Breakpoints: `condition BPNUMBER EXPRESSION`.
- Editing variables at runtime with `gdb`:
  - `set var VARIABLE = VALUE` assigns `VALUE` to `VARIABLE`.
  - `set {int}0x1234 = 4` writes `4` (as an integer) to the memory address `0x1234`.
- Disassembling code: `disassemble FUNCTION` prints the assembly for a function named `FUNCTION`.

Further Reading

- GDB manual

- More on breakpoints
- KDbg, a GUI for gdb
- rr, tool that can record and replay program execution!