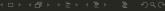
# How to Use a Debugger Focusing on GDB

Richard Morrill

Fordham University CS Society

Thursday, Novemeber 7th 2019





#### Intro

- WTF is a debugger anyways?
- Why would I want one?
- How the heck do I set all this crap up?



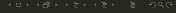


#### Setup

Let's get all the tech support out of the way.

- Open Instructions Doc: http://bit.ly/GDBSetup
- Write / Compile a Simple C/C++ Program
- Run It, Make Sure it Works
- Run \$ gdb <your executable name>
- Run (gdb)<sup>1</sup> run





<sup>&</sup>lt;sup>1</sup>This means you run the command inside GDB

# The Problem a Debugger Solves

```
int doSomething(int* arr, int len) {
    for (int i = 0; i < len; ++i) {
        if (arr[i] < 4) {</pre>
             arr[i] = complexFunction(arr[i]);
    for(int i = 0; i < len; ++i) {
           (simpleFunction(arr[i])) {
             return arr[i];
        } else {
             arr[i] = doSomething(arr, len);
```

How to Use a Debugger

#### The Problem a Debugger Solves, Cont.

- I want to know where the program is crashing
- So I do this:

```
#include <stdio.h?
```





#### The Problem a Debugger Solves, Cont.

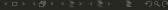
- I want to know where the program is crashing
- So I do this:

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

And this...

```
printf("some random variable: %d", var);
```





#### The Problem a Debugger Solves, Cont.

- I want to know where the program is crashing
- So I do this:

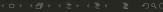
```
#include <stdio.h>
```

And this...

```
printf("some random variable: %d", var);
```

- My code is full of print statements
- My terminal is flooded with info

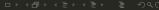




#### What I Want to Do

- Stop my program anywhere I want
- See what all the variables are
- See how they got that way
- If it crashes, see what was going on when it crashed
- Not need to delete all those stupid print statements afterward



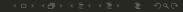


# What a Debugger Can Do for Me

- Stop my program: breakpoints, watchpoints, on exceptions
- See values of all variables (and registers!)
- Follow <u>call stack</u> back up<sup>2</sup>
- Play "what if?"



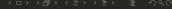
<sup>&</sup>lt;sup>2</sup>Underlined terms are things not everybody will know, please ask me to explain them.



# Why Are We Doing This the Hard Way?

Builds character

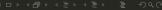




# Why Are We Doing This the Hard Way?

- Builds character
- Forces you to be intentional
- Cross-platform skills
- Many features hidden / not available in GUI





#### Essential Commands

I forget them constantly and so will you

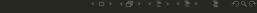
- Open this: http://bit.ly/GDBCheatSheet
- Essential Commands:
  - \$ gdb EXECUTABLE start debugging
  - (gdb) run ARGS... run your program inside debugger
  - (gdb) break [FUNCTION|FILE:LINE] set breakpoint
  - (gdb) watch EXPRESSION set watchpoint
  - (gdb) step single step
  - (gdb) print EXPRESSION print value
  - (gdb) kill start over
  - (gdb) quit exit
  - (gdb) continue start running again
  - (gdb) [up|down] traverse the call stack
  - (gdb) clear [FUNCTION|FILE:LINE] remove watchpoint / breakpoint



# Time for Some Real Debugging

- Navigate to the Basic\_Usage folder
- Compile with \$ gcc hasfunctions.c -lm -o hasfunctions<sup>3</sup>
- Run it with 2 int arguments<sup>4</sup>





<sup>&</sup>lt;sup>3</sup>Anybody need me to explain what the arguments are doing?

<sup>&</sup>lt;sup>4</sup>Anybody not know what that means?

# Time for Some Real Debugging

- Navigate to the Basic\_Usage folder
- Compile with \$ gcc hasfunctions.c -lm -o hasfunctions<sup>3</sup>
- Run it with 2 int arguments<sup>4</sup>
- 90% chance you'll get a segfault
- Run it in gdb





<sup>&</sup>lt;sup>3</sup>Anybody need me to explain what the arguments are doing?

<sup>&</sup>lt;sup>4</sup>Anybody not know what that means?

# Set a Breakpoint

```
11   int* doesSomethingElse(double first, int count) {
        int* myArr = malloc(count * sizeof(int));
        double trail;
        for(int i = 0; i < count; ++i) {
            myArr[i] = trail - first * 2;
            trail = myArr[i] + first * 3 - floor(first);
        }
        recount;
        recou
```

hasfunctions.c

I want to stop every time the loop goes around



# Set a Breakpoint

```
int* doesSomethingElse(double first, int count) {
    int* myArr = malloc(count * sizeof(int));
    double trail:
   for(int i = 0; i < count; ++i) {
        myArr[i] = trail - first * 2;
        trail = myArr[i] + first * 3 - floor(first);
```

#### hasfunctions.c

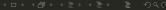
- I want to stop every time the loop goes around
- Answer: (gdb) break hasfunctions.c:15
- Breakpoints stop before executing the line



### More About Breakpoints

- On x86 CPUs, very fast
- Great when you need to know what is going on at any given point in code

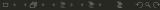




### More About Breakpoints

- On x86 CPUs, very fast
- Great when you need to know what is going on at any given point in code
- Not so great when you don't actually know where that point is
- Potential Solutions:

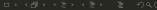




#### More About Breakpoints

- On x86 CPUs, very fast
- Great when you need to know what is going on at any given point in code
- Not so great when you don't actually know where that point is
- Potential Solutions:
  - Set a shit-ton of breakpoints
  - Use accessor / mutator functions
  - Watchpoints





#### Watchpoints

When you have no idea when, where, or how

- Watchpoints trigger whenever a given variable or expression<sup>5</sup> changes
- Less intuitive to set than a breakpoint
- More technical limitations than breakpoints
- Can be implemented in both hardware and software, use software with (gdb) set can-use-hw-watchpoints 06



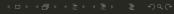


<sup>&</sup>lt;sup>5</sup>This is most relevant if you know how to use pointers, who does?

<sup>&</sup>lt;sup>6</sup>You might need to do this if using WSL

- Open / compile / run hasglobals.c<sup>7</sup>
- Try setting a watchpoint on globalvar

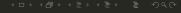




- Open / compile / run hasglobals.c<sup>7</sup>
- Try setting a watchpoint on globalvar (gdb) watch globalVar
- Try setting a watchpoint on globalPtr



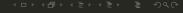
<sup>&</sup>lt;sup>7</sup>Of course, watchpoints work on local vars too, but less clear to explain



- Open / compile / run hasglobals.c<sup>7</sup>
- Try setting a watchpoint on globalvar (gdb) watch globalVar
- Try setting a watchpoint on globalPtr
- What if we want to see when the memory it's pointing to changes instead?



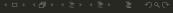
<sup>&</sup>lt;sup>7</sup>Of course, watchpoints work on local vars too, but less clear to explain



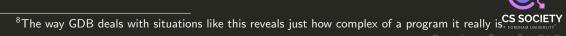
- Open / compile / run hasglobals.c<sup>7</sup>
- Try setting a watchpoint on globalvar (gdb) watch globalVar
- Try setting a watchpoint on globalPtr
- What if we want to see when the memory it's pointing to changes instead?
  - (gdb) watch \*globalVar



<sup>&</sup>lt;sup>7</sup>Of course, watchpoints work on local vars too, but less clear to explain



- It's very obvious when you want to watch a global
- What about if you want to watch a local?

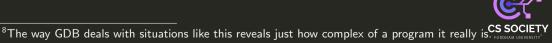


- It's very obvious when you want to watch a global
- What about if you want to watch a local? Breakpoint in scope,
   (gdb) watch var
- Try watching a value that doesn't exist yet: globalPtr[2]



 $<sup>^8</sup>$ The way GDB deals with situations like this reveals just how complex of a program it really is. ECSS

- It's very obvious when you want to watch a global
- What about if you want to watch a local? Breakpoint in scope,
   (gdb) watch var
- Try watching a value that doesn't exist yet: globalPtr[2]
- GDB is fine with breakpoints that can't be read yet<sup>8</sup>



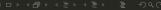
- It's very obvious when you want to watch a global
- What about if you want to watch a local? Breakpoint in scope,
   (gdb) watch var
- Try watching a value that doesn't exist yet: globalPtr[2]
- GDB is fine with breakpoints that can't be read yet<sup>8</sup>
- However, there are limitations:
  - If you change the value of globalPtr, you'll no longer be watching the same location
  - If you manually set a watchpoint on a memory location pointed to by globalPtr, you won't follow changes to globalPtr

<sup>&</sup>lt;sup>8</sup>The way GDB deals with situations like this reveals just how complex of a program it really is. FORDINAL UNIVE

### Explore the Call Stack

- Go back to hasfunctions.c
- Break inside goodNumber()
- Pretend you don't know which function called it, how would you find out?

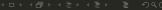




# Explore the Call Stack

- Go back to hasfunctions.c
- Break inside goodNumber()
- Pretend you don't know which function called it, how would you find out? –
  (gdb) bt

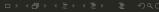




# Explore the Call Stack

- Go back to hasfunctions.c
- Break inside goodNumber()
- Pretend you don't know which function called it, how would you find out? –
   (gdb) bt
- Play around with (gdb) up
- You can access local scope in any one of the <u>frames</u>





#### **GDB Extras**

- (gdb) list \*\$pc Show current location in code
- (gdb) save breakpoints FILENAME Save all breakpoints to a file for later
- (gdb) source FILENAME Recover saved breakpoints
- (gbd) info registers See register values

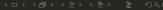




<sup>&</sup>lt;sup>9</sup>For some reason this means breakpoints AND watchpoints.

- Find out what your computer is really doing
- Try (gdb) dissasemble main





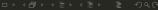
- Find out what your computer is really doing
- Try (gdb) dissasemble main
- Probably looks like garbage, try: (gdb) set dissasembly-flavor intel





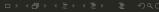
- Find out what your computer is really doing
- Try (gdb) dissasemble main
- Probably looks like garbage, try: (gdb) set dissasembly-flavor intel
- Break somewhere in main, try disassembling main again
  - You can see where the program counter is





- Find out what your computer is really doing
- Try (gdb) dissasemble main
- Probably looks like garbage, try: (gdb) set dissasembly-flavor intel
- Break somewhere in main, try disassembling main again
  - You can see where the program counter is
- Disassembly is useful when you don't have the source code but you need to figure out how something works.
- May be a topic for a future event.

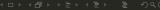




#### Syscalls

- Requests your program makes to the OS
- Knowing which calls it's making and when can be very useful when dealing with files & networking
- Go to Syscalls, compile and run makesSyscalls.cpp

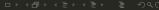




#### Syscalls

- Requests your program makes to the OS
- Knowing which calls it's making and when can be very useful when dealing with files & networking
- Go to Syscalls, compile and run makesSyscalls.cpp
- Wow, look at all that gibberish!
- Pay attention to openat, read, write
- Notice how there's only one read call, why?

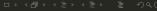




#### Syscalls

- Requests your program makes to the OS
- Knowing which calls it's making and when can be very useful when dealing with files & networking
- Go to Syscalls, compile and run makesSyscalls.cpp
- Wow, look at all that gibberish!
- Pay attention to openat, read, write
- Notice how there's only one read call, why? Compiler and/or OS optimization





# Debugging in VsCode

- On linux, VsCode's debugger actually uses GDB behind the scenes
- Watch as I set it up and use it





# Thanks for Coming

Next Week: Research Meeting



