

Question to Audience:

How much time (as a percentage of their day) do you think programmers spend **debugging** their code?

Social: [@MichaelShah](https://twitter.com/MichaelShah)
Web: mshah.io
Courses: courses.mshah.io
YouTube: www.youtube.com/c/MikeShah

11:00-12:00, Mon, 12th September 2022

60 minutes | Introductory Audience



20
22



Back To Basics

Debugging

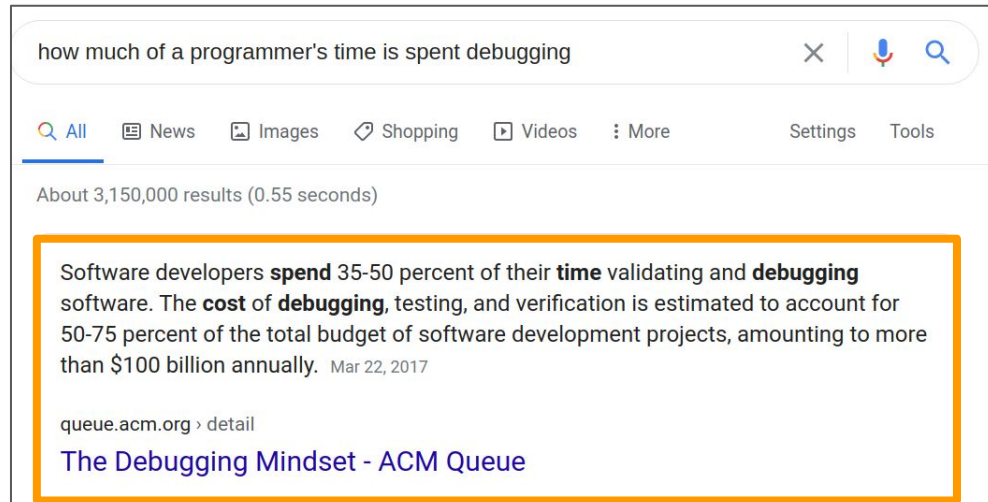
Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)
YouTube: [www.youtube.com/c/MikeShah](#)

11:00-12:00, Mon, 12th September 2022

60 minutes | Introductory Audience

The Answer

- Estimate from 2017 is between 35-50% of our time, and the cost is 50-75% of the budget!
 - (as of 2017 from ACM Queue)
 - <https://queue.acm.org/detail.cfm?id=3068754#:~:text=Software%20developers%20spend%2035%2D50,more%20than%20%24100%20billion%20annually.>
- Sounds like we need to learn some tools to save us time!



Abstract

I always tell my students, the debugger is your 'get out of jail free card' when working on a project. I say the same thing to professionals, debuggers are your 'get out of free jail card'. The reality is that programmers spend the majority of their time debugging as opposed to writing new code. Unfortunately many programmers do not learn how to use a debugger, or otherwise how they should approach debugging. In this talk I am going to show you how to debug C++ code, starting from the very basics and then demonstrating how a debugger like GDB can be used to help you track errors in CPU code. Attendees at this talk will learn names of debugging techniques (e.g. delta debugging), and I will demonstrate several debugging tools (stepping through code, capturing backtraces, conditional breakpoints, scripting, and even time traveling!) to demonstrate the power of debuggers. This is a beginner friendly talk where we are going to start from the beginning, but I suspect I may show a trick or two that folks with prior experience will appreciate.

Please do not redistribute slides without prior permission.

Goal(s) for today

1. Understand Debugging Strategies
2. Introduction to GDB

What you'll learn today -- the metaphor (1/3)

- For those familiar with the board game monopoly [\[wiki\]](#), there's a part of the game where you can 'go to jail'
 - Generally, that's a bad thing in the game
 - But if you know how to use a debugger, ... (next slide)



What you'll learn today -- the metaphor (2/3)

- For those familiar with the board game monopoly [\[wiki\]](#), there's a part of the game where you can 'go to jail'
 - Generally, that's a bad thing in the game
 - But if you know how to use a debugger, **it's kind of like having one of these**





where you can

- Generally, that's a bad thing in the game
- But if you know how to use a debugger, it's kind of like having one of these
- **In fact, if you know how to use your debugging tools, it's like having a lot of these 'get out of jail free' cards, that help you get out of tricky situations!**

An Introduction to getting yourself out of trouble

- Learn some tools (focusing on a debugger) to find bugs and fix bugs
 - And I would also argue that learning a debugger is just another useful tool for helping you how to understand how a program executes.

Learn how to get yourself out of trouble by learning essential debugging skills.

I always tell my students, that understanding how to debug is one way to understand how a program works. Debuggers allow you to see inside of a program, understand the workflow the state of a program. If you're on a new project for instance, what better way to get familiar with the product than to step through it one line or function at a time. And of course, the main reason to learn debugging is because some estimate we spend up to 90% of our programming time debugging -- and now writing new code. Debugging is an essential skill, and you should learn at least the basics in this course!

<https://courses.mshah.io/> (Shameless self-promotion!)

Your Tour Guide for Today

by Mike Shah (he/him)

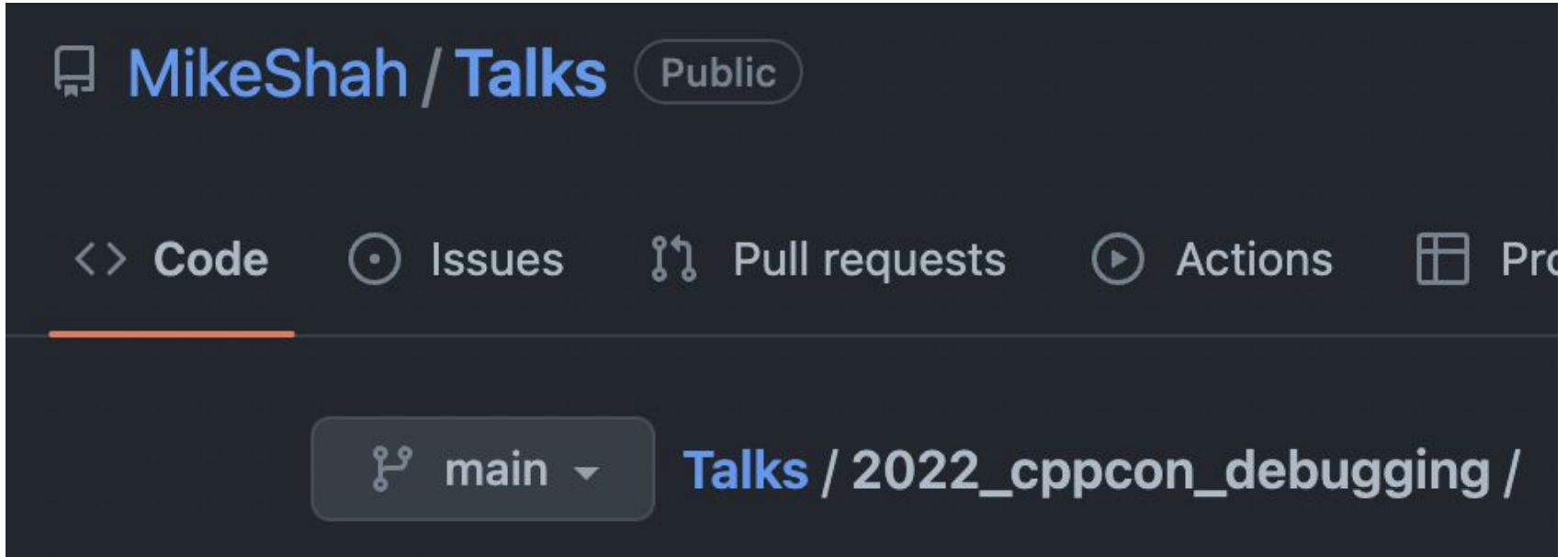
- **Associate Teaching Professor** at Northeastern University in Boston, Massachusetts.
 - I teach courses in computer systems, computer graphics, and game engine development.
 - My **research** in program analysis is related to **performance** building static/dynamic analysis and software visualization tools.
- I do **consulting** and technical training on modern C++, Concurrency, OpenGL, and Vulkan projects
 - (Usually graphics or games related)
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of **computer graphics**, visualization, concurrency, and parallelism.
- Contact information and more on: www.mshah.io
- More online training at courses.mshah.io



Code for the talk

- Located here:

https://github.com/MikeShah/Talks/tree/main/2022_cppcon_debugging



What is a bug?

A good place to start

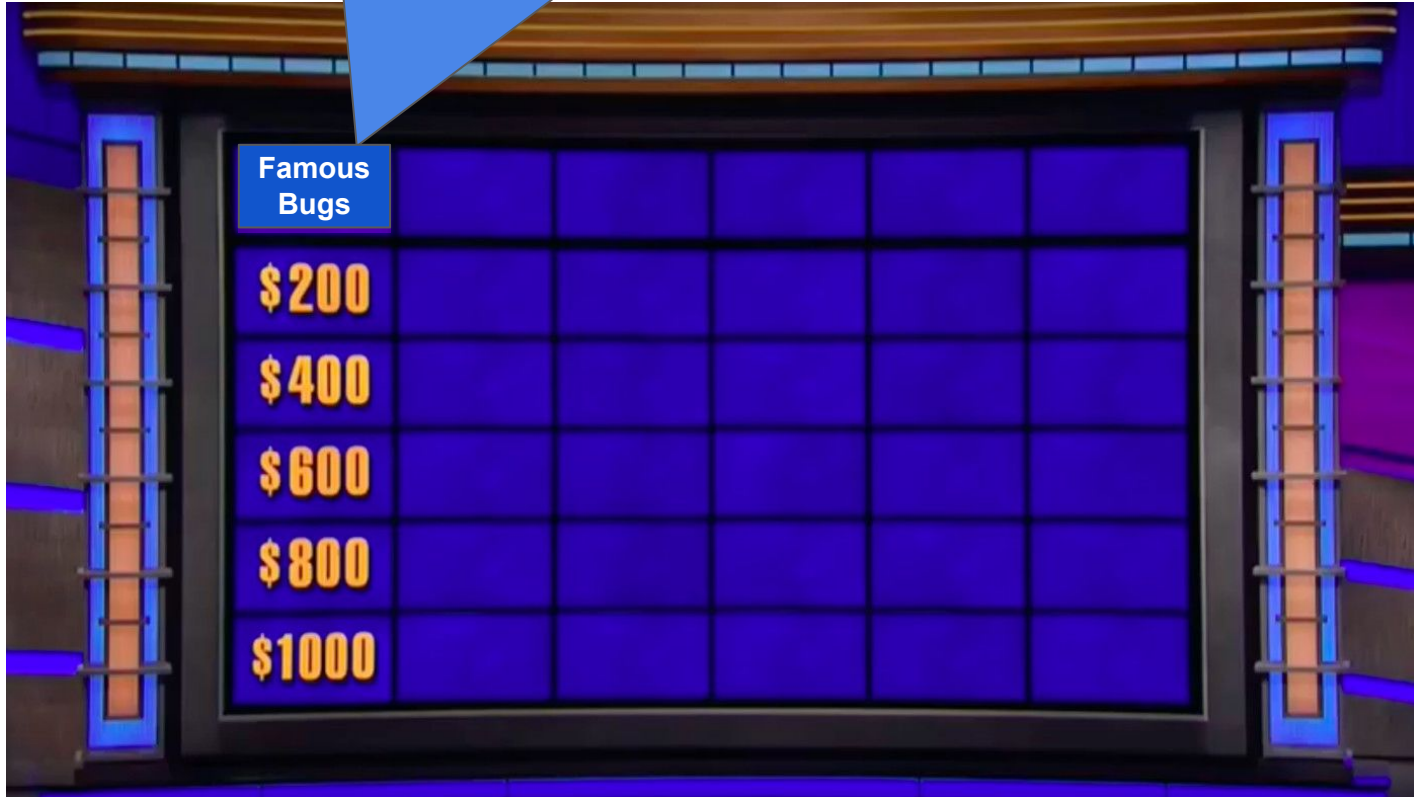
Some images today from the wonderful movie 'A Bug's Life' by Disney Pixar.
Apologies for any spoilers! It is a great movie! :)



What is a Software Bug?

- A [software bug](#) is a *defect* in the logic, correctness, or performance of a software system
 - It is a *fault* that we want to fix it to match our expectations or a technical specification.
 - **(logic)** Programs that compiles, but does not do at runtime do what the developer expects
 - **(correctness)** Program executes path as expected but produces the wrong result
 - **(performance)** Performance bugs may be dependent on workload on your system or an external system (e.g. a server)
 - **(nondeterministic correctness and logic)** Heisenbugs for example are bugs that occur in concurrent code and are sporadically observable
- Software bugs can sometimes go undetected for long periods of time and be difficult to find, depending on the class of the bug
 - Let's take a moment to look at some infamous software bugs... (next slide)

Infamous Software Bugs

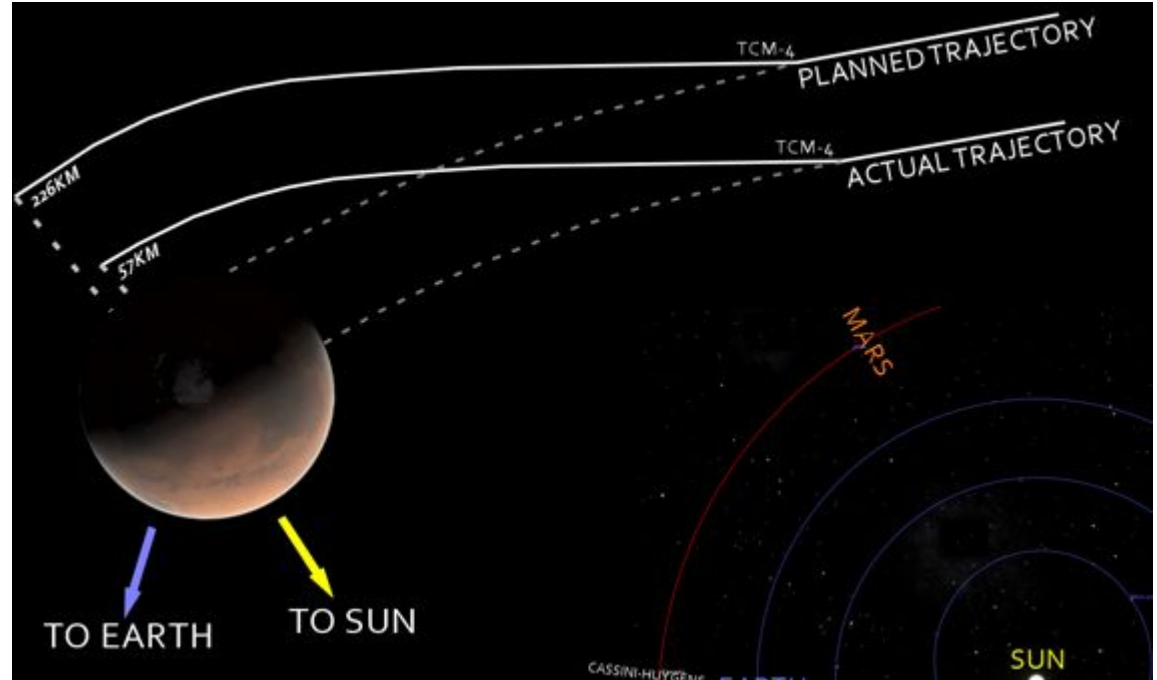


This image is from the American game show "Jeopardy" in which contestants answer questions in the form of a question to earn money

- 9/9
- 0800 Antman started
- 1000 " stopped - antman ✓
- 1300 (032) MP-MC 1.58247000
2.130476415 (2) 4.615925059(-2)
- (033) PRO 2 2.130476415
convd 2.130676415
- Relays 6-2 in 033 failed special speed test
in relay 11.000 test.
- Relays changed
- 1100 Started Cosine Tape (Sine check)
- 1525 Started Multi-Adder Test.
- 1545
- Relay #70 Panel F
(moth) in relay.
- First actual case of bug being found.
- 1630 Antman started.
- 1700 closed down.

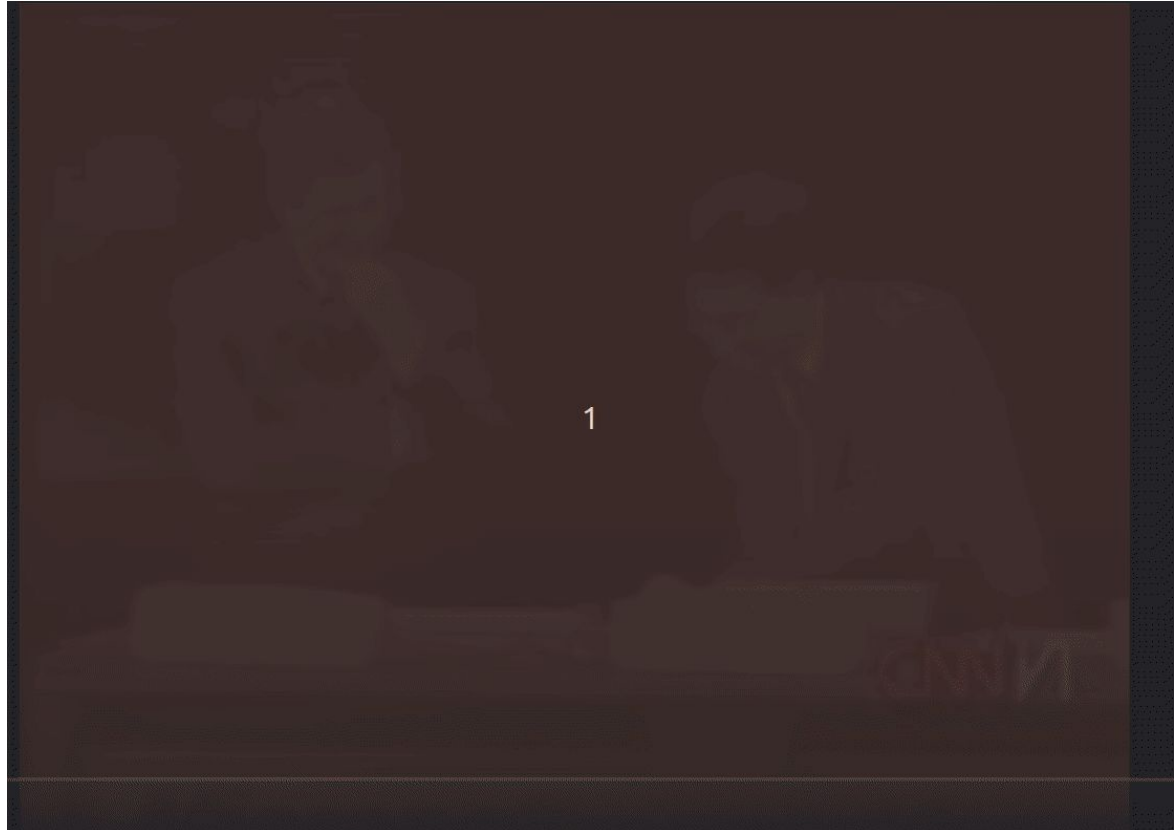
Mars Climate Orbiter - 1998

- Did they mean to put the units in feet or meters?
 - Software calculations were in meters...
 - Team controlling entered parameters in imperial units
- The probe made an error of about 100km and was destroyed
- [Link to story](#)



Win 98 Blue Screen of Death (~1998)

- This next one is a correctness bug you can see in action!
 - This happened in front of a live audience
 - <https://www.youtube.com/watch?v=yeUyxjLhAxU> (41 seconds)
 - (The gentleman to the right was not a programmer but in marketing, and later Chief Marketing Officer)
- [Link to story](#)



Win 98 Blue Screen of Death (~1998)

And I don't mean to embarrassed this gentleman on the right -- we know developing software can be tricky!

- This next one is a correctness bug you can see in action!
 - This happened in front of a live audience
 - <https://www.youtube.com/watch?v=yeUyxjLhAxU> (41 seconds)
 - (The gentleman to the right was not a programmer but in marketing, and later Chief Marketing Officer)
- [Link to story](#)



Y2K Bug - 1999

- Software developers did not think ahead about code that would last into the new millennium, thus abbreviating 1999 to “99”
 - Banks worried ‘00’ would be interpreted as ‘1900’ and mess up interest rate calculations
 - Media thought there would be disasters (and the bug was real), though we survived.
- [Link to story](#)



More bugs (Costly bugs!) [[source](#)]

- 1962
 - Mariner 1 Spacecraft nearly crashes due to a software error (\$18 million 1962 dollars)
 - Missing 'hyphen' in data transmitted back was 1 of 2 major errors [[source](#)]
- 1988
 - The Morris worm spreads wildly out of control causing an estimated \$100 million in damages
 - Error was in the worms 'replication logic' [[source](#)]
- 1994
 - Intel's popular pentium processor had a math error in the fdiv operation costing them \$475 million in recalls. [[source](#)]
- 2010
 - Bitcoin Hack lost about 850,000 bitcoins [[source](#)]
- And many more...(the list doesn't start stop at 2010...)

More bugs (Costly bugs!) [[source](#)]

- 1962
 - Mariner 1 Space Probe
 - Missing 1962 dollars) [source]
- 1988
 - The Morris worm
 - Error was \$100 million in damages
- 1994
 - Intel's popular Pentium processor
 - Error was costing them \$475 million
- 2010
 - Bitcoin Hack
- And many more...(the list doesn't stop at 2010)



Why are we creating bugs?

What's the difficulty?



Are you kiddin'?

Why is it hard to get software correct? (1/2)

- **Question to the audience:** Why is it hard to write correct software? Your thoughts?

Why is it hard to get software correct? (2/2)

- **Question to the audience:** Why is it hard to write correct software? Your thoughts?
 - (Some of my thoughts)
 - Software changes frequently!
 - The C++ Standard language specification is 1841 pages [\[link\]](#).
 - It's probably hard to use every feature correctly (or for performance)
 - Lots of programmer and managers work on a project
 - Programmers rely on building a mental model (some approximation) of the software to reason about behavior
 - Likely this mental model will differ amongst some number of programmers and managers
 - Pressure between meeting tight deadlines and economic costs
 - (i.e., technical debt accrues and make sit hard to write correct software)
 - Poor documentation of APIs -- and sometimes APIs are broken!
 - Lack of testing (unit tests, behavior tests, etc.)
 - Unanticipated inputs (bad user input) or unexpected system events (network down)
 - The reality is, we are humans and will make mistakes!

Today's topic unfortunately however...is
too much of a mystery



Today's topic unfortunately however...is

Today's Goal

Learn some debugging techniques

“Although computer science education devotes a lot of time to teaching algorithms and fundamentals, it appears that not much of this time is spent applying them to general problems. Debugging is not taught as a specific course in universities. Despite decades of literature suggesting such courses be taught, no strong models exist for teaching debugging.” [[ACM Queue](#) The Debugging Mindset 2017]

Still not enough debugging taught in University

- I dedicate lectures to debugging in university-- but even one full 100 minute lecture is not enough!
 - (Even better, some courses I sprinkle in debugging tools throughout the course)
- But I cannot remember a single lecture during my time in university on debugging

Schedule/Road Map

The following is our tentative syllabus for the course, changes should be expected throughout the semester. I will announce in class, piazza, or through e-mail any major changes.

Week	Date	Lecture and Readings	Assignments	Note(s)
1	Tuesday - September 14, 2021	Module 1 - Course Introduction, C++, and Revision Systems(Git)	A1 Released (Due Sept. 23 Anywhere on Earth(AOE)) Lab 1 Out(Due. Sept 23 AOE)	Welcome back to class! (Note: First in-person class is the 14th and 16th respectively for each section)
2	Tuesday - September 21, 2021	Module 2 - C++ Object Oriented Programming and how to Structure a C++ Project	A2 Out (Due Oct. 5) Lab 2 out (Due Sept. 30 AOE)	-- --
3	Tuesday - September 28, 2021	Module 3 - Design Considerations and Design Patterns 1	Lab 3 out (Due. Oct. 7 AOE)	-- --
4	Tuesday - October 05, 2021	Module 4 - Software Development Lifecycle	A3 Out (Due. Oct. 20 Anywhere on Earth) Lab 4 Out (Due Oct. 14 AOE)	Mike out of town for Thursday. Thursday's section will watch Tuesday video.
5	Tuesday - October 12, 2021	Module 5 - Debugging	Lab 5 out (Due. Oct. 21 AOE)	-- --

Some wisdom from Dr. Greg Law

<https://www.youtube.com/watch?v=QOo27EmHuu0&feature=youtu.be&t=6>

Debugging - our dirty secret

Most of a developer's time is spent debugging.

What happened?



Debugging versus testing

- Debugging is closely related to testing, and both are necessary skills to learn as software engineers
 - Testing means we are checking for the presence of a bug (given an input, test an expected output)
 - Debugging is the process of removing an observed fault in our software
 - We might test again after debugging to confirm the bug has been isolated
 - And likely we may add a unit test to a test suite after debugging
- **Please see the many talks on testing at the conference throughout the week**

ONSITE & ONLINE ACCESS Back to Basics Testing + Add to Schedule	ONLINE ACCESS ONLY Software Testing Group Q&A In Gather Town + Add to Schedule	ONSITE & ONLINE ACCESS "It's A Bug Hunt" Armor Plate Your Unit Tests + Add to Schedule	ONSITE & ONLINE ACCESS How to Use Dependency Injection to Write Maintainable Software + Add to Schedule
Idioms & Techniques 09:00 - 10:00 Amir Kirsh	Debugging & Logging & Testing Gather Town 12:30 - 13:30 Dave Steffen Phil Nash Amir Kirsh Fedor Pikus Ben Saks	Debugging & Logging & Testing 09:00 - 10:00 Dave Steffen	Debugging & Logging & Testing 16:45 - 17:45 Francesco Zoffoli

Debugging Techniques and Strategies

This is interactive--see if you can spot the bug!



#1 Scan and Look Debugging



Common Strategy - Scan and look (1/4)

- If you're familiar with the software, sometimes you can just 'find it'
 - This is called the 'scan and look' strategy for bug finding
 - Let's try it out below

Common Strategy - Scan and look (2/4)

- If you're familiar with the software, sometimes you can just 'find it'
 - This is called the 'scan and look' strategy for bug finding
 - Let's try it out below

Try to find the bug!

mike@mike-MS-7B17: ~/C-SoftwareEngineering/5

File Edit View Search Terminal Help

```
1 // printfdebug.cpp
2 // g++ -std=c++17 printfdebug.cpp -o printfdebug
3 #include <iostream>
4
5 int main(){
6
7     int PI = 3.1415;
8
9     std::cout << "The value of pi is:" << PI << std::endl;
10
11
12     return 0;
13 }
```

(Need a hint--next slide)

Common Strategy - Scan and look (3/4)

- If you're familiar with the software, sometimes you can just 'find it'
 - This is called the 'scan and look' strategy for bug finding
 - Let's try it out below

Try to find the bug!

mike@mike-MS-7B17: ~/C-SoftwareEngineering/5

```
1 // printfdebug.cpp
2 // g++ -std=c++17 printfdebug.cpp -o printfdebug
3 #include <iostream>
4
5 int main(){
6
7     int PI = 3.1415;
8
9     std::cout << "The value of pi is:" << PI << std::endl;
10
11
12     return 0;
13 }
```

```
mike:5$ g++ -std=c++17 printfdebug.cpp -o printfdebug
mike:5$ ./printfdebug
The value of pi is:3
mike:5$
```

Common

The bug has been spotted!

- If you're Logical error/typo by the programmer. can just 'find it'
 - This
 - Let's

Did not provide the correct type.

The lesson here--even if code compiles, it does not imply correctness!

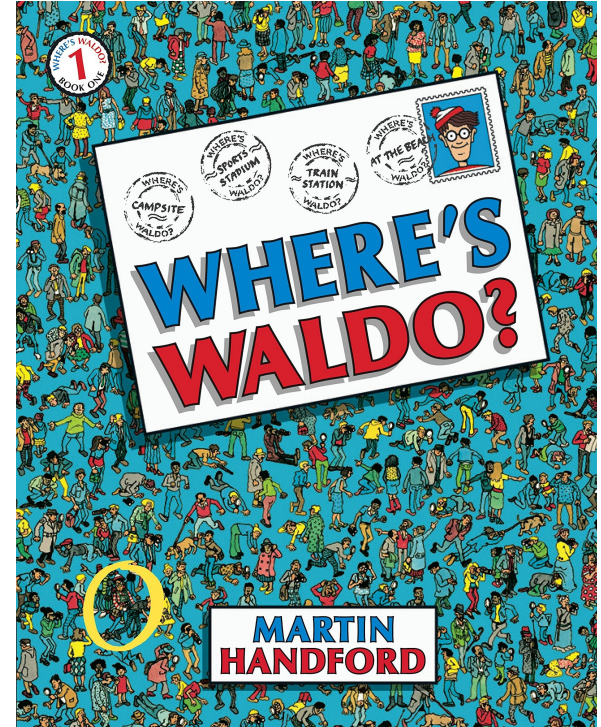
File Edit View Search Terminal Help

```
1 // printfdebug.cpp
2 // g++ -std=c++17 printfdebug.cpp -o printfdebug
3 #include <iostream>
4
5 int main(){
6
7     int PI = 3.1415;
8
9     std::cout << "The value of pi is:" << PI << std::endl;
10
11
12     return 0;
13 }
```

```
mike:5$ ./printfdebug
The value of pi is:3
mike:5$
```

Tradeoffs - Scan and Look (1/2)

- Pros
 - Anyone can use this strategy, and there are no external tools needed.
- Cons
 - Not reliable, in some sense you are guessing where the error is
 - (See Where's Waldo image on the right)
 - This strategy typically does not scale well
 - e.g. Code you did not write is hard to scan
 - e.g. This strategy is likely to be tedious on even small projects (< 1,000 lines of code (LOC)).



Where's Waldo is a children's book where you try to find the main character
<https://images-na.ssl-images-amazon.com/images/I/A1aulg-l7WL.jpg>

Tradeoffs - Scan and Look (2/2)

- Pros

- Anyone can find Waldo with the right tools needed

- Cons

- Not reliable, error is

- (See V)

- This strategy

- e.g. C

- e.g. This strategy is likely to be tedious on even small projects (< 1,000 lines of code (LOC)).

The good news is, we have a tool that can help us having to do this strategy automatically for us.

The compiler can help!



Scan and look (with the compilers help) (1/2)

- Using the scan and look strategy can be exhausting
 - So we can improve this solution by using our compiler (it sees all of our code!)
 - `-Wall` and `-Wextra` are flags sent to the compiler that will help catch some of these types of errors.
 - Hmm, looks like the error was not caught with `g++` though...

Try to find the bug!

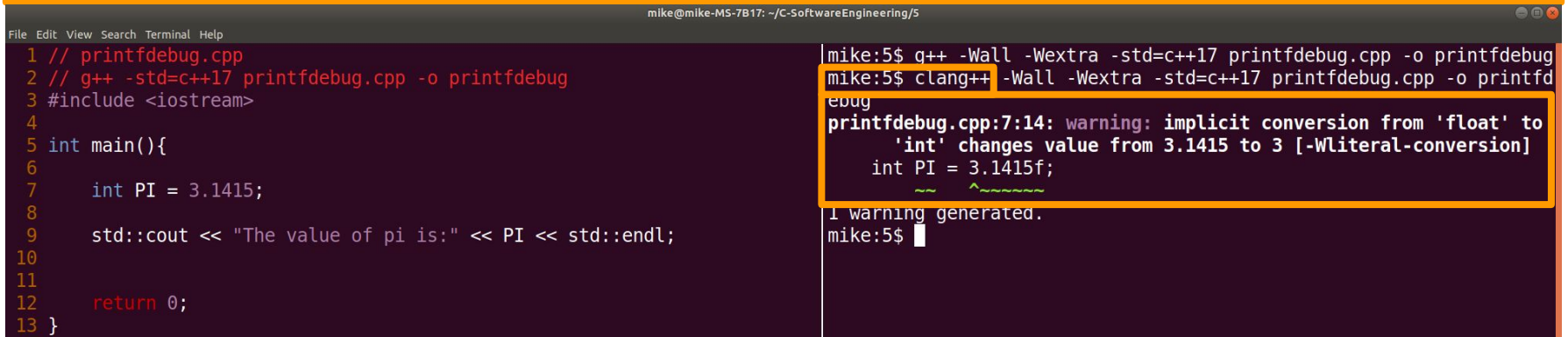
```
mike@mike-MS-7B17: ~/C-SoftwareEngineering/5
File Edit View Search Terminal Help
1 // printfdebug.cpp
2 // g++ -std=c++17 printfdebug.cpp -o printfdebug
3 #include <iostream>
4
5 int main(){
6
7     int PI = 3.1415;
8
9     std::cout << "The value of pi is:" << PI << std::endl;
10
11
12     return 0;
13 }

mike:5$ g++ -Wall -Wextra -std=c++17 printfdebug.cpp -o printfdebug
mike:5$
```

Scan and look (with the compilers help) (2/2)

- Using the scan and look strategy can be exhausting
 - So we can improve this solution by using our compiler (it sees all of our code!)
 - `-Wall` and `-Wextra` are flags sent to the compiler that will help catch some of these types of errors.
 - Hmm, looks like the error was not caught with `g++` though...

Tip: Switching between **clang++** and **g++** (and vice versa) may sometimes report different warnings



```
mike@mike-M5-7B17: ~/C-SoftwareEngineering/5
File Edit View Search Terminal Help
1 // printfdebug.cpp
2 // g++ -std=c++17 printfdebug.cpp -o printfdebug
3 #include <iostream>
4
5 int main(){
6
7     int PI = 3.1415;
8
9     std::cout << "The value of pi is:" << PI << std::endl;
10
11
12     return 0;
13 }

mike:5$ g++ -Wall -Wextra -std=c++17 printfdebug.cpp -o printfdebug
mike:5$ clang++ -Wall -Wextra -std=c++17 printfdebug.cpp -o printfdebug
printfdebug.cpp:7:14: warning: implicit conversion from 'float' to
'int' changes value from 3.1415 to 3 [-Wliteral-conversion]
    int PI = 3.1415f;
               ^~~~~~
1 warning generated.
mike:5$
```


Tradeoffs - Scan and look (with the compilers help)

- Pros

- Our compiler scales--meaning it can report on errors at compile-time for large programs
- Over time, compilers tend to get better at finding more errors

```
mike:5$ g++ -Wall -Wextra -std=c++17 printfdebug.cpp -o printfdebug
mike:5$ clang++ -Wall -Wextra -std=c++17 printfdebug.cpp -o printfdebug
printfdebug.cpp:7:14: warning: implicit conversion from 'float' to
'int' changes value from 3.1415 to 3 [-Wliteral-conversion]
    int PI = 3.1415f;
               ^~~~~
1 warning generated.
mike:5$
```

- Cons

- Only works at compile-time (no bugs found at runtime)
- Only types of warnings we can fix are what the compiler reports on.
- What if we don't have the source code?
 - (i.e., libraries that we link in)
 - We cannot fix those warnings!

#2 printf debugging

A technique for helping us debug and retrieve values at run-time



Common Strategy - printf debugging (1/5)

- `printf` is the 'C' function for displaying text on the console
 - (The equivalent in C++ is `std::cout`)
- The idea of printf debugging is that we can print a value at a particular point in our source code to discover the state of our program.
 - We then have to recompile the program and execute it to observe the change

Common Strategy - printf debugging (2/5)

- `printf` is the 'C' function for displaying text on the console
 - (The equivalent in C++ is `std::cout`)
- The idea of printf debugging is that we can print a value at a particular point in our source code to discover the state of our program.

Try to find the bug!

```
File Edit View Search Terminal Help
1 // printfdebug2.cpp
2 // g++ -std=c++17 printfdebug2.cpp -o printfdebug2
3 #include <iostream>
4 #include <stdlib.h>
5
6 int square(int a){
7     return a;
8 }
9
10 int main(){
11
12     while(1){
13         if(square(5)==25){
14             exit(1);
15         }
16     }
17
18     std::cout << "Exiting program\n";
19
20     return 0;
21 }
```

```
mike@mike-MS-7B17: ~/C-SoftwareEngineering/5
mike:5$ g++ -Wall -Wextra -std=c++17 printfdebug2.cpp -o printfdebug2
mike:5$ ./printfdebug2
```

No warnings this time

But see if you can spot the bug (Don't say anything yet!)

Common Strategy - printf debugging

- `printf` is the 'C' function for printing text to the screen
 - (The equivalent in C++ is `std::cout`)
- The idea of printf debugging is to insert `printf` statements in our source code to discover the state of the program.

Depending on how much code I put on the screen--this bug can be harder to find!

Let's try to help ourselves out with some output (i.e. `printf`) statements

(Bug shown on the next slide)

in

find the bug!

```
File Edit View Search Terminal Help
1 // printfdebug2.cpp
2 // g++ -std=c++17 printfdebug2.cpp -o printfdebug2
3 #include <iostream>
4 #include <stdlib.h>
5
6 int square(int a){
7     return a;
8 }
9
10 int main(){
11
12     while(1){
13         if(square(5)==25){
14             exit(1);
15         }
16     }
17
18     std::cout << "Exiting program\n";
19
20     return 0;
21 }
```

```
mike:5$ g++ -Wall -Wextra -std=c++17 printfdebug2.cpp -o printfdebug2
mike:5$ ./printfdebug2
```

No warnings this time

But see if you can spot the bug (Don't say anything yet!)

Common Strate

- `printf` is the 'C'
 - (The equivalent)
- The idea of `printf`

Some well placed output statements *anywhere state can change* (i.e. a value can be generated or a variable mutated) reveal the value of `square(5)`.

We observe the incorrect value, and confirm we never enter the branch and see 'output 2'

```
File Edit View Search Terminal Help
1 // printfdebug3.cpp
2 // g++ -std=c++17 printfdebug3
3 #include <iostream>
4 #include <stdlib.h>
5
6 int square(int a){
7     return a;
8 }
9
10 int main(){
11
12     while(1){
13
14         std::cout << "output 1: " << square(5) << std::endl;
15         if(square(5)==25){
16             std::cout << "output 2: " << square(5) << std::endl;
17             exit(1);
18         }
19         std::cout << "output 3: " << square(5) << std::endl;
20     }
21
22     std::cout << "Exiting program\n";
23
24     return 0;
25 }
```

[illegible]

Common Strate

- `printf` is the ‘C’
 - (The equivalent)
- The idea of `printf`

oops, an error in our
functions return
value--should be $(a*a)$

```
1 // printfdebug3.cpp
2 // g++ -std=c++17 printfdebug3.cpp -o printfdebug3
3 #include <iostream>
4 #include <stdlib.h>
5
6 int square(int a){
7     return a;
8 }
9
10 int main(){
11
12     while(1){
13
14         std::cout << "output 1: " << square(5) << std::endl;
15         if(square(5)==25){
16             std::cout << "output 2: " << square(5) << std::endl;
17             exit(1);
18         }
19         std::cout << "output 3: " << square(5) << std::endl;
20     }
21
22     std::cout << "Exiting program\n";
23
24     return 0;
25 }
```

[illegible]

Tradeoffs - printf Debugging

- Pros

- Can help narrow down where bugs occur
- You can observe values at run-time
- You get an idea of where execution is.
- Can 'pretty print' or otherwise format your data output.

- Cons

- You may need to make many educated guesses in long running programs
- You are also modifying the source code directly, and need to remember to remove your output statements
- It requires you to rebuild your software
 - Recompilation for every small change can be expensive in terms of time
- It requires you to build additional infrastructure which may or may not be needed
 - Meaning: Not every object has or needs to be printed out, but you will need to see a textual representation of that object

#3 Delta Debugging

(A technique to help us narrow our search space for where a bug occurs)



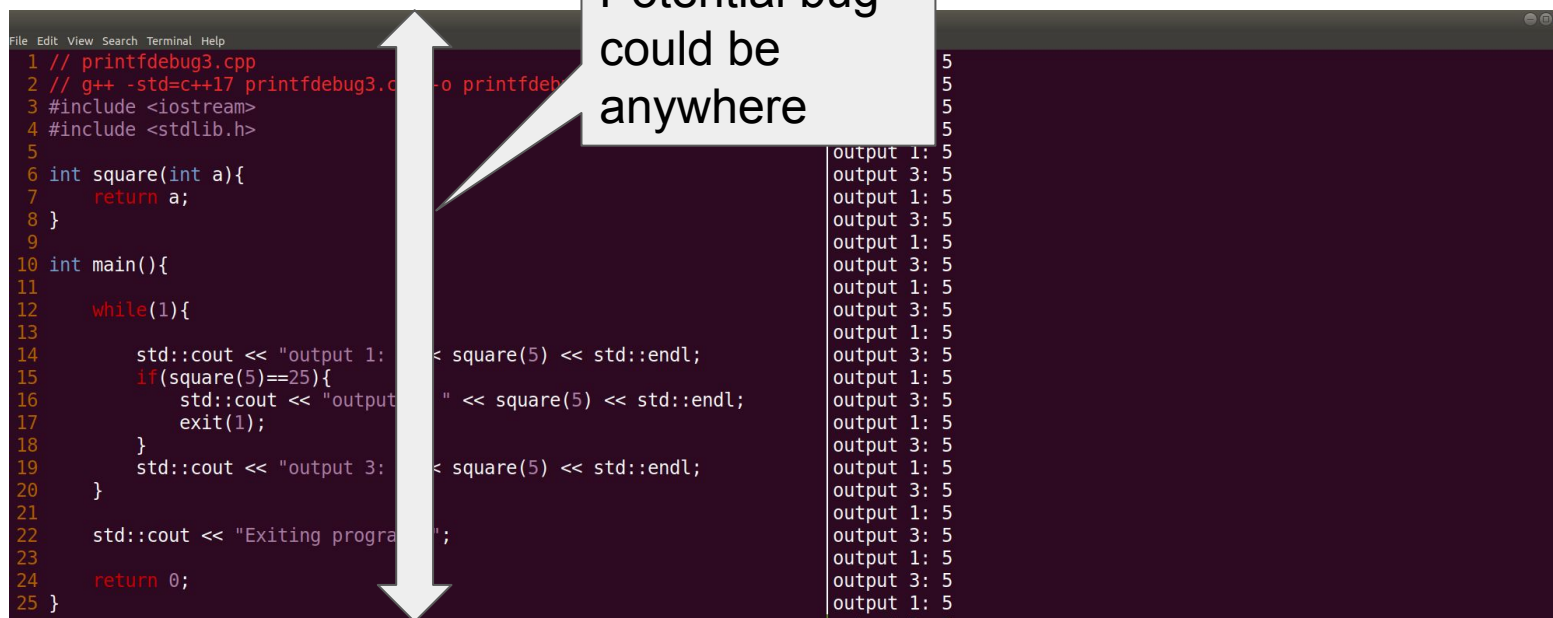
Strategy for debugging - Delta Debugging (1/3)

- With the printf debugging strategy, you are trying to shrink your delta of where an error could occur.
 - This is called Delta debugging

[illegible]

Strategy for debugging - Delta Debugging (2/3)

- With the printf debugging strategy, you are trying to shrink your delta of where an error could occur.
 - This is called Delta debugging

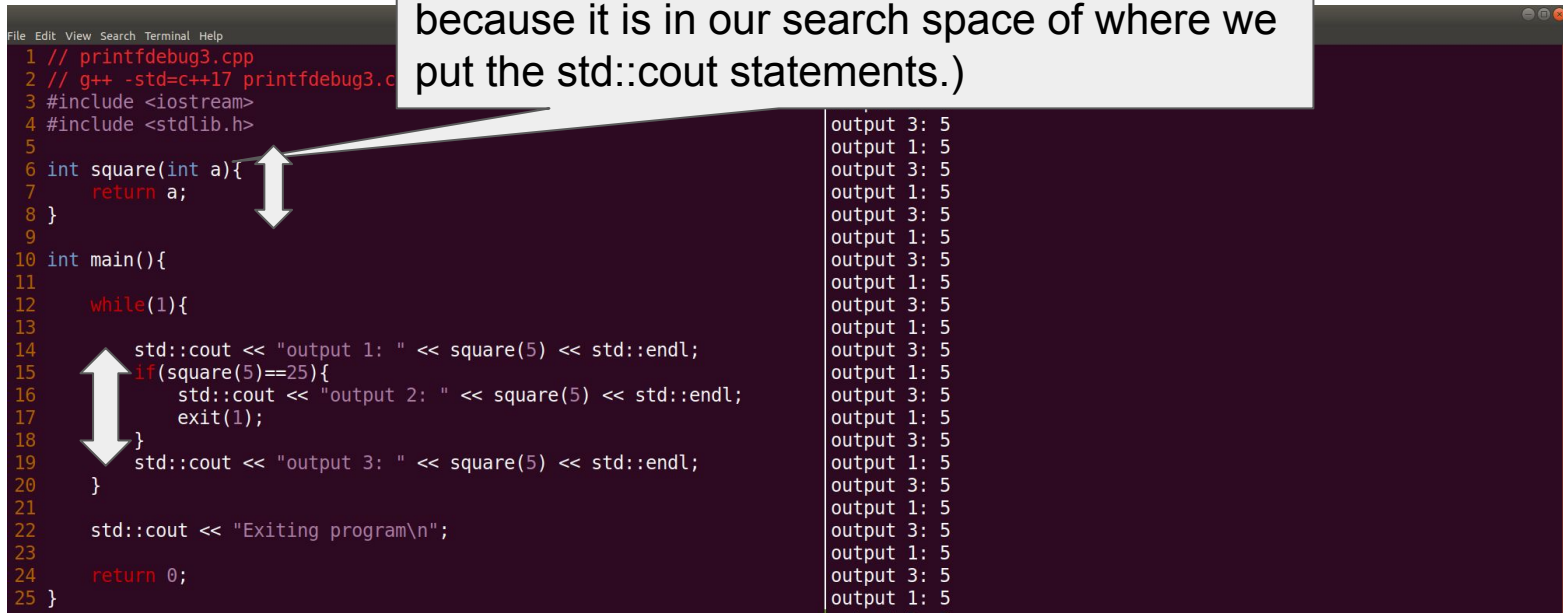


Strategy for debugging - Delta Debugging (3/3)

- With the printf debugging technique, you can find the delta of where an error could occur
 - This is called Delta

Bug is somewhere in this range

(Note square function is included in our delta because it is in our search space of where we put the std::cout statements.)



The screenshot shows a C++ program in a terminal window. The program defines a `square` function and a `main` function. The `main` function has a `while(1)` loop that prints the output of `square(5)` three times. The first two prints are followed by a conditional check `if(square(5)==25)`, which prints the output of `square(5)` again if it is true. The program then prints "Exiting program\n" and returns 0.

```
1 // printfdebug3.cpp
2 // g++ -std=c++17 printfdebug3.c
3 #include <iostream>
4 #include <stdlib.h>
5
6 int square(int a){
7     return a;
8 }
9
10 int main(){
11     while(1){
12
13         std::cout << "output 1: " << square(5) << std::endl;
14         if(square(5)==25){
15             std::cout << "output 2: " << square(5) << std::endl;
16             exit(1);
17         }
18         std::cout << "output 3: " << square(5) << std::endl;
19     }
20 }
21
22 std::cout << "Exiting program\n";
23
24 return 0;
25 }
```

The output of the program is shown on the right side of the terminal window. It consists of a repeating sequence of three lines: "output 3: 5", "output 1: 5", and "output 3: 5". This sequence is repeated multiple times, indicating that the program is stuck in a loop.

Tradeoffs - Delta Debugging

- Pros

- Can help narrow down search space (thus saving you time)
- Even if you don't have the source code--you can still isolate where the error *may* be occurring.

- Cons

- It relies on you to have knowledge of your software, and pick a good *delta* (i.e. search space)
 - You may have to spend more time picking a delta.

```
mike@mike-M5-7B17
1 // printfdebug3.cpp
2 // g++ -std=c++17 printfdebug3.cpp -o printfdebug3
3 #include <iostream>
4 #include <stdlib.h>
5
6 int square(int a){
7     return a;
8 }
9
10 int main(){
11
12     while(1){
13
14         std::cout << "output 1: " << square(5) << std::endl;
15         if(square(5)==25){
16             std::cout << "output 2: " << square(5) << std::endl;
17             exit(1);
18         }
19         std::cout << "output 3: " << square(5) << std::endl;
20     }
21
22     std::cout << "Exiting program\n";
23
24     return 0;
25 }
```

#4 printf debugging revisited

Improving our printf debugging using our programming language



printf Debugging - Slightly enhanced

- There are some programming techniques you can use to help you find and report bugs
- The C++ language allows us to utilize something called 'the preprocessor' which does textual replacement before compiling our code
 - The **preprocessor** allows us to:
 - Choose to conditionally have our printf statements show up at compile-time
 - Write a Macro (a textual replacement function)
 - (Some of you are shuddering at the word 'Macro' -- but everything has tradeoffs!)

The video player displays a hand-drawn diagram of the C/C++ compilation process. The diagram shows the flow from Source.cpp to the final executable, including the preprocessor, compiler, assembler, linker, and dynamic linker. It also shows the use of flags like -I, -L, and -l.

In 54 Minutes, Understand the whole C and C++ compilation process

Watch this to understand our compiler and where macros come into place! [In 54 Minutes, Understand the whole C and C++ compilation process](#)

printf Debugging using the preprocessor

- Here's an example where I use `#ifdef` to check if a symbol has been defined.
 - Observe in the first compilation below there is no output
 - Observe in the second compilation:
 - I pass in `-D _DEBUG` to the compiler
 - You will observe the output is different based on the conditional compilation in the preprocessor.

```
mike@system76-pc: ~/C-SoftwareEngineering/5
File Edit View Search Terminal Help
1 // preprocessor.cpp
2 // g++ -std=c++17 preprocessor.cpp -o preprocessor
3
4 #include <iostream>
5
6
7 int main(){
8     float PI = 3.1415;
9
10    #ifdef _DEBUG
11        std::cout << "Pi is:" << PI << "\n";
12    #endif
13
14    return 0;
15 }
16 }
```

11,17 All

```
1 mike@system76-pc:~/C-SoftwareEngineering/5$ g++ -std=c++17 preprocessor.cpp -o preprocessor
mike@system76-pc:~/C-SoftwareEngineering/5$ ./preprocessor
2 mike@system76-pc:~/C-SoftwareEngineering/5$ g++ -D _DEBUG -std=c++17 preprocessor.cpp -o preprocessor
mike@system76-pc:~/C-SoftwareEngineering/5$ ./preprocessor
Pi is:3.1415
mike@system76-pc:~/C-SoftwareEngineering/5$
```

printf Debugging using macros

- Here is an example of writing a Macro function
 - You can wrap statements or functions with Macros
 - This saves typing, and provides an opportunity to report information like file and line information.

```
File Edit View Search Terminal Help
1 // macro.cpp
2 // g++ -std=c++17 macro.cpp -o macro
3
4 #include <iostream>
5
6 // Note the '\\' allows us to use multiple lines
7 #define PRINT(x,line,file) \
8     std::cout << file << ":" << line << ": "; \
9     std::cout << "value is: " << (x) << "\n";
10
11
12 int main(){
13
14     float PI = 3.1415;
15
16     PRINT(PI, __LINE__, __FILE__)
17
18     return 0;
19 }
```

9,28 All

```
mike@system76-pc:~/C-SoftwareEngineering/5$ g++ -std=c++17 macro.
cpp -o macro
mike@system76-pc:~/C-SoftwareEngineering/5$ ./macro
macro.cpp:16: value is: 3.1415
mike@system76-pc:~/C-SoftwareEngineering/5$
```

Tradeoffs - Using Preprocessor to Debug

- Pros

- Can make the code slightly cleaner
- Having error macros available may encourage quick error checking
 - This could be enforced in a style guide
- Can more easily turn macros on and off

- Cons

- Still requires source modification
- Macros can quickly expand generate lots of code which may be hard to debug
- Using Macros will increase compilation time
- Macros left in the final build can be expensive for build time, or accidentally logging sensitive information
- Can add 'clutter' to the programmers mental model of how code actually executes
- Some debuggers will not easily expand Macros (i.e. need to use -ggdb3 option with gdb)

```
mike@system76-pc:~/C-SoftwareEngineering/5
1 // preprocessor.cpp
2 // g++ -std=c++17 preprocessor.cpp -o preprocessor
3
4 #include <iostream>
5
6
7 int main(){
8
9     float PI = 3.1415;
10
11     #ifdef DEBUG
12         std::cout << "PI is:" << PI << "\n";
13     #endif
14
15     return 0;
16 }
```

11,17 All

```
mike@system76-pc:~/C-SoftwareEngineering/5$ g++ -std=c++17 preprocessor.cpp -o preprocessor
mike@system76-pc:~/C-SoftwareEngineering/5$ ./preprocessor
mike@system76-pc:~/C-SoftwareEngineering/5$ g++ -D_DEBUG -std=c++17 preprocessor.cpp -o preprocessor
mike@system76-pc:~/C-SoftwareEngineering/5$ ./preprocessor
PI is:3.1415
mike@system76-pc:~/C-SoftwareEngineering/5$
```

```
mike@system76-pc:~/C-SoftwareEngineering/5
1 // macro.cpp
2 // g++ -std=c++17 macro.cpp -o macro
3
4 #include <iostream>
5
6 // Note the '\ ' allows us to use multiple lines
7 #define PRINT(x,line,file) \
8     std::cout << file << ":" << line << " : " << x << "\n"; \
9     std::cout << "value is: " << (x) << "\n";
10
11
12 int main(){
13
14     float PI = 3.1415;
15
16     PRINT(PI, __LINE__, __FILE__)
17
18     return 0;
19 }
```

9,28 All

```
mike@system76-pc:~/C-SoftwareEngineering/5$ g++ -std=c++17 macro.cpp -o macro
mike@system76-pc:~/C-SoftwareEngineering/5$ ./macro
macro.cpp:16: value is: 3.1415
mike@system76-pc:~/C-SoftwareEngineering/5$
```


Breaking Old Habits

- Here are some techniques we have seen:
 - 1. scan and look
 - 2. utilizing our compilers
 - 3. delta debugging
 - 4. printf debugging
 - 5. printf with conditional compilation
 - 6. printf with macros
- However, while in practice they are valid--I want to break some old habits
- ****I want your first resource to be to use an interactive debugger the next time you encounter a bug.****
 - (i.e. not scatter little print statements in your program)

Interactive Debuggers

Tools that allow introspection into code at run-time e.g., GDB



Yes....you will have a part of this -- debuggers save you time!

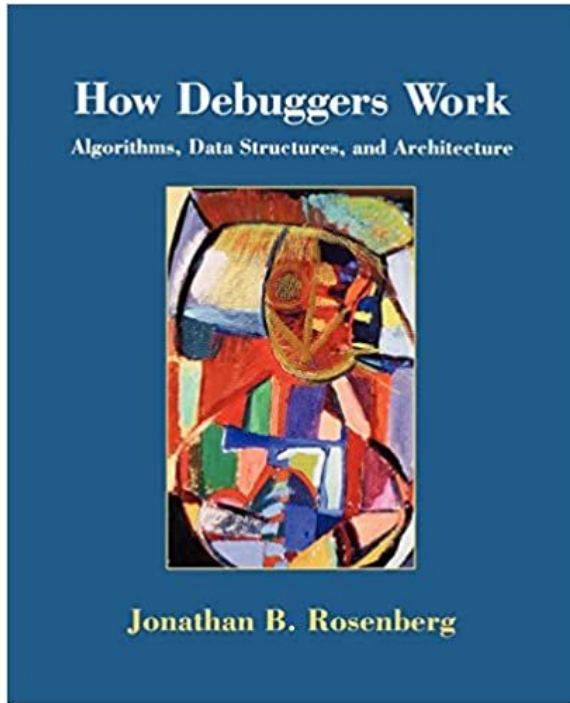
Interactive Debugger

- Interactive debuggers allows us to inspect our program without source modification
 - (They can sometimes however be a form of dynamic binary instrumentation)
- Today I want to show you how to use an interactive debugger so you can resolve your C++ bugs
 - Using GDB (or the debugger associated with your operating system/IDE) will be your first line of defense!



How Debuggers Work

- Debuggers work by attaching to a running process
 - (This means we debug at run-time)
 - Typically debuggers use special system calls in the operating system to handle events that take place within the specific process they are attached to.
- For linux users, you can investigate [ptrace](#)
 - For other operating systems there is an equivalent system call you can further look at.



Compiling with Debugging Symbols to help GDB

- Adding debugging symbols when compiling your program, provides more information to the debuggers when you execute your program
 - Information like source file and line number become more clear
 - Typically you can recover symbols for variable and function names in your source files as well
 - (Extra debugging information is stored typically in a 'symbol table' or other auxiliary data structure)
- Takeaway:
 - When compiling, use '-g' to get debugging symbols
 - (There are a few other options like : -ggdb or -g1, -g2, -g3)
 - (-g0 provides no debug information)

Running your program with GDB

- Most often, when you execute your program, you are going to execute it within gdb.
 - GDB provides you a command-line interface to interactively explore and execute your program
- Starting GDB
 - From within GDB you can type 'run' or 'r' to start executing your program
 - Or alternatively 'start' which will pause your program (using a breakpoint) at the main function.

GDB Live Code

Sample code available at:

https://github.com/MikeShah/Talks/tree/main/2022_cppcon_debugging

GNU Debugger GDB (1/2)

- I am going to teach you how to use the GNU GDB Debugger today
- This is a free debugger available on Windows, linux, and Mac

```
GDB(1)                                GNU Development Tools                                GDB(1)

NAME
    gdb - The GNU Debugger

SYNOPSIS
    gdb [-help] [-nh] [-nx] [-q] [-batch] [-cd=dir] [-f] [-b bps]
        [-tty=dev] [-s symfile] [-e prog] [-se prog] [-c core] [-p procID]
        [-x cmds] [-d dir] [prog|prog procID|prog core]

DESCRIPTION
    The purpose of a debugger such as GDB is to allow you to see what is
    going on "inside" another program while it executes -- or what another
    program was doing at the moment it crashed.

    GDB can do four main kinds of things (plus other things in support of
    these) to help you catch bugs in the act:

    · Start your program, specifying anything that might affect its
      behavior.
```

GNU Debugger GDB (2/2)

- I am going to show how to use GDB Debugger
- This is a free available linux, and

You can use whatever debugger you like, but I will show examples in GDB for you to follow along with.

Most IDEs have the same functionality and methodology that I will show, perhaps a different workflow

```
GDB(1)
[-f] [-b bps]
[-c core] [-p procID]
re]

w you to see what is
tes -- or what another

things in support of

might affect its
```

Let's dive in

- I want to spend some time looking at a simple piece of code
- Starting with a simple example is a good way to start!
 - (Here's what we'll cover)
 - Compiling with debug symbols
 - Running GDB
 - Starting a program
 - Executing each line one at a time
 - Listing the source code
 - Setting some breakpoints

mike:2\$


1

Round 2 -GDB TUI (Text User Interface)

- Many folks do not know--GDB provides a textual user interface
- You can use *Control-x 1 (or Control-x 2)* to enable it.
 - Note: It can take a little practice to switch into the TUI Mode
 - I prefer to just launch with tui: `gdb . /prog --tui`
- *Ctrl-x o* will cycle you through the windows in the tui mode
- You can additionally type 'list' if you want to see the source code you are looking at.
 - *list linenumber* (e.g. list 10)

Breakpoints and stepping through code


- The basic workflow when debugging is to set a 'breakpoint' 'br' at a specific function or line in your program.
- This pauses execution until you decide to resume.
 - You can
 - 'continue' - Continues execution until the next breakpoint
 - 'step' - step to the next line of code that will execute
 - 'next' - execute the next instruction
- After you set a breakpoint you can:
 - display them: 'info breakpoints'
 - remove them 'delete *breakpoint 1*' (e.g. deletes first breakpoint)
 - *save breakpoints filename*
 - *source filename* (loads the breakpoints)



```
mike@system76-pc:~/C-SoftwareEngineering/5$
```

print

- When you are at a breakpoint, you want to observe a value
 - From now on--you do not have to litter your code with 'std::cout' statements.
- The 'print' command allows you to do that.
 - print variable
 - (or in hex: print/x variable)
 - And you can also 'dereference if it's a pointer'
 - e.g., print *variable
 - And you can also print the address of a variable
 - print &variable



```
mike@system76-pc:~/C-SoftwareEngineering/5$
```



Watchpoints

- You can use a 'watch' to interrupt your program and set a break every time that a variables value is modified.
 - e.g.,
 - 'watch i' in a loop
 - (Then use 'continue' to continue execution)

```
mike@system76-pc:~/C-SoftwareEngineering/5$ gdb ./gdbexample
```

Conditional Breakpoints


- If you want to monitor variables in a loop, you can set conditional breakpoints that watch for a particular condition
- e.g.
 - `break main.cpp:20 if i > 5`
 - `break main.cpp:20 if i > j`

A screenshot of a terminal window with a black background and white text. The prompt 'mike@system76-pc:~/C-SoftwareEngineering/5\$' is visible at the top left. A mouse cursor is positioned in the center of the terminal area.

```
mike@system76-pc:~/C-SoftwareEngineering/5$
```

Backtrace (retrieving the call stack)

- Segmentation faults can be one of the more common errors you encounter, and often you'll have to changes of state over time.
- You can use the 'backtrace' command to see 'how' or otherwise what functions were called to get you in that location.
 - You can use the command 'bt' to review where the program crashed by retrieving a program stack
 - Then 'info args' or 'info frame' to



```
mike@system76-pc:~/C-SoftwareEngineering/5$
```

GDB - Attach to a running process

- (using gdb2.cpp)
- Graphics applications like we have been working on run in infinite loops
- If you have already started executing a program, you can attach a debugger to it
 - `ps -elf | grep program_name`
 - look for the Process ID (PID)
 - `sudo gdb attach {PID number}`
 - Usually you'll need root privileges
 - Helpful hint: Use 'finish' to execute until a function is finished in case you are in some library of code when you attach to a process.
 - (Or otherwise use 'up' to move up the call stack)

```
mike@system76-pc:~/C-SoftwareEngineering/5$ ./gdbexample2
mike@system76-pc:~/C-SoftwareEngineering/5$ ps -elf | grep gdbexample2
```

Slightly More Advanced Example (time travel)

- (gdb3.cpp example)
- More advanced debuggers allow for 'time travel' and reverse debugging
 - target record-full
 - next
 - reverse-next
 - reverse-step

```
mike:2022_cppcon_debugging$ gdb --tui --silent ./prog
```

1

Slightly More Advanced Example (polymorphism)

- (gdb3.cpp example)
- Many IDEs do not support some of the more advanced features
- How do we know object types?
 - `whatis object_name`
- How do we know how an object is behaving?
 - `info vtbl object_name`

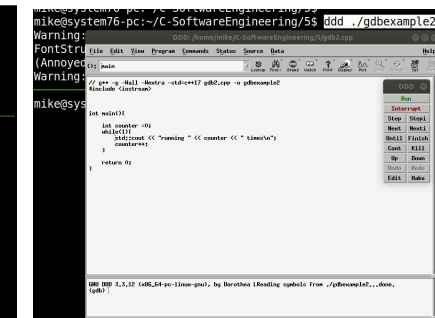
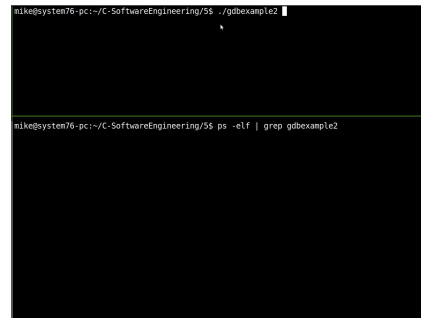
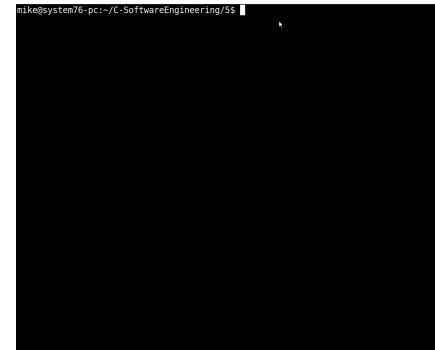
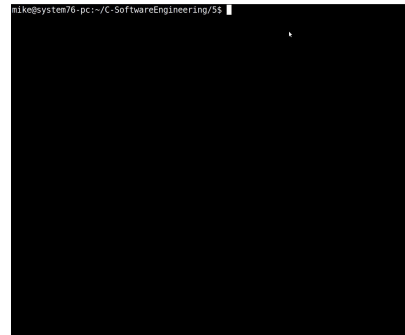
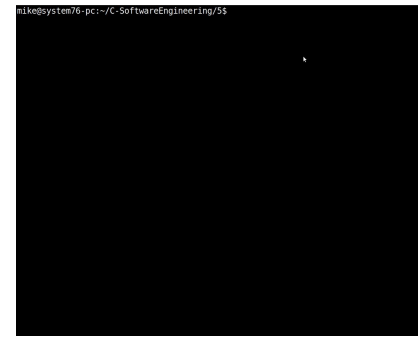
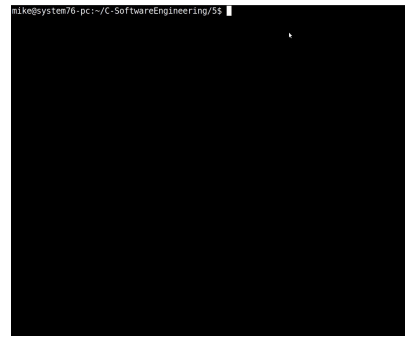
```
File Edit View Search Terminal Help
16 Dog(){
17     std::cout << "Dog Constructor";
18 }
19 void action(){
20     std::cout << "Dog::action()";
21 }
22 };
23
24 int main(){
25
26     Animal* genericAnimal;
27     genericAnimal = new Dog;
28     // What will happen here?
29     genericAnimal->action();
30     // How about after this
31     delete genericAnimal;
32     // Try 'whatis' in gdb
33     genericAnimal = new Animal;
34     genericAnimal->action();
35
36     return 0;
37 }

gdb3.cpp
24 int main(){
25
26     Animal* genericAnimal;
27     genericAnimal = new Dog;
28     // What will happen here?
29     genericAnimal->action();
30     // How about after this
31     delete genericAnimal;
32     // Try 'whatis' in gdb
33     genericAnimal = new Animal;
34     genericAnimal->action();
35
36     return 0;
37 }
38
39
40
41
42

native process 28637 In: main L29 PC: 0x55555554b6e
[0]: <error: Cannot access memory at address 0x89485ed18949ed31>
[1]: <error: Cannot access memory at address 0x89485ed18949ed39>
[2]: <error: Cannot access memory at address 0x89485ed18949ed41>
(gdb) n
(gdb) info vtbl genericAnimal
vtable for 'Animal' @ 0x555555755d10 (subobject @ 0x555555768e70):
[0]: 0x55555554d90 <Dog::~Dog()>
[1]: 0x55555554dba <Dog::~Dog()>
[2]: 0x55555554d6e <Dog::action()>
(gdb)
```

Debugging Summary

- Debugging Techniques
 - Use your debugging tools!
 - Compile with ' -g' while developing
 - Treat warnings as errors that need to be fixed (-Werror).
 - Use -Wall and -Wextra
 - Use two compilers
 - GDB will help you solve your problems much quicker than guessing and recompiling.



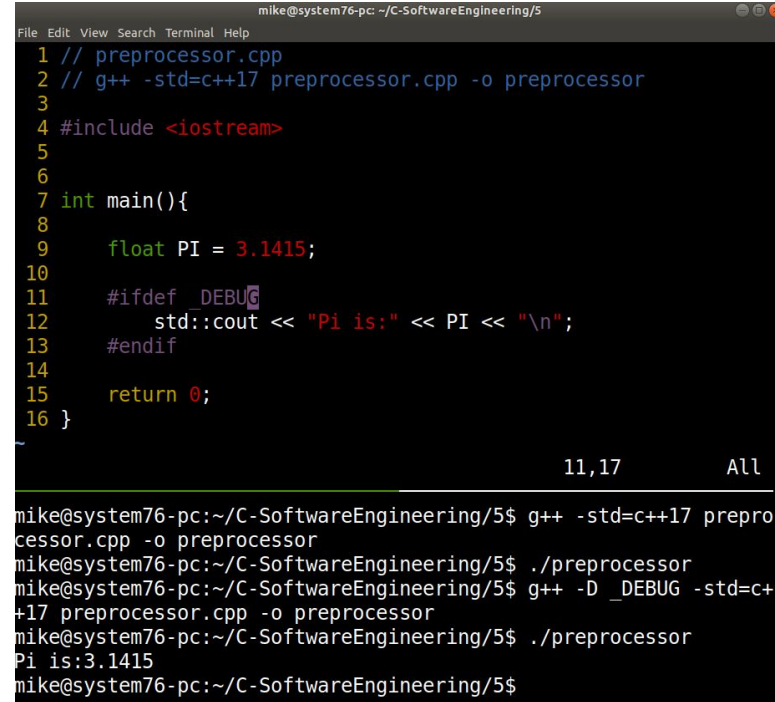
Debug and Release Builds

Other considerations to be careful of when distributing software to the masses



Debug and release builds (1/2)

- Recall that we did define a symbol previously `_DEBUG` (or sometimes `DEBUG`)
- Just a note that we typically call this a 'debug build'
- When we do not include debug symbols, we call that the release build.
 - **Question to audience:** Why might we not want to give to consumers a 'debug build'?

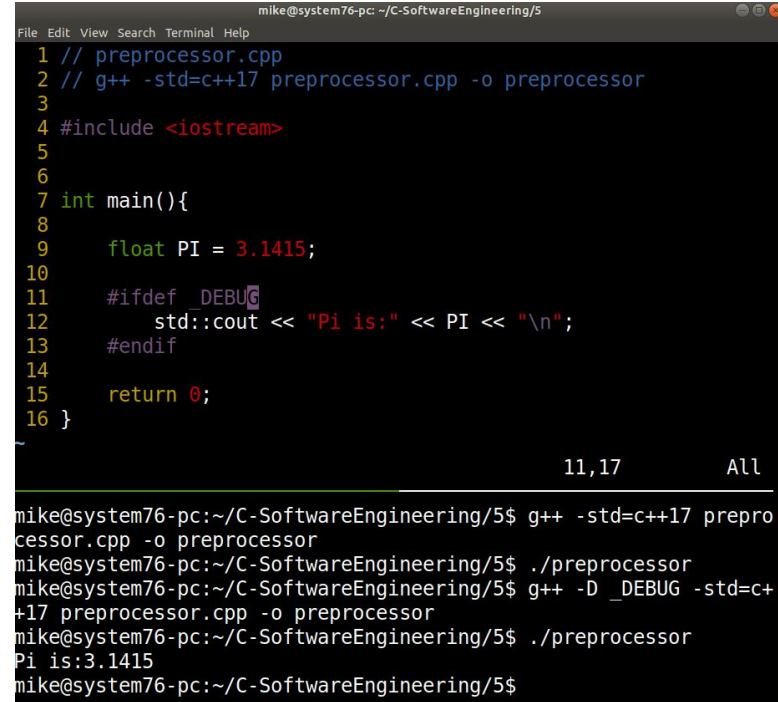


```
mike@system76-pc: ~/C-SoftwareEngineering/5
File Edit View Search Terminal Help
1 // preprocessor.cpp
2 // g++ -std=c++17 preprocessor.cpp -o preprocessor
3
4 #include <iostream>
5
6
7 int main(){
8
9     float PI = 3.1415;
10
11     #ifdef _DEBUG
12         std::cout << "Pi is:" << PI << "\n";
13     #endif
14
15     return 0;
16 }

11,17 All
mike@system76-pc:~/C-SoftwareEngineering/5$ g++ -std=c++17 preprocessor.cpp -o preprocessor
mike@system76-pc:~/C-SoftwareEngineering/5$ ./preprocessor
mike@system76-pc:~/C-SoftwareEngineering/5$ g++ -D_DEBUG -std=c++17 preprocessor.cpp -o preprocessor
mike@system76-pc:~/C-SoftwareEngineering/5$ ./preprocessor
Pi is:3.1415
mike@system76-pc:~/C-SoftwareEngineering/5$
```

Debug and release builds (2/2)

- Recall that we did define a symbol previously `_DEBUG` (or sometimes `DEBUG`)
- Just a note that we typically call this a 'debug build'
- When we do not include debug symbols, we call that the release build.
 - **Question to audience:** Why might we not want to give to consumers a 'debug build'
 - Hackers can see extra information!
 - Note: You can use various tools (strip on linux for example) to remove debugging information.



```
mike@system76-pc: ~/C-SoftwareEngineering/5
File Edit View Search Terminal Help
1 // preprocessor.cpp
2 // g++ -std=c++17 preprocessor.cpp -o preprocessor
3
4 #include <iostream>
5
6
7 int main(){
8
9     float PI = 3.1415;
10
11     #ifdef _DEBUG
12         std::cout << "Pi is:" << PI << "\n";
13     #endif
14
15     return 0;
16 }

11,17 All
mike@system76-pc:~/C-SoftwareEngineering/5$ g++ -std=c++17 preprocessor.cpp -o preprocessor
mike@system76-pc:~/C-SoftwareEngineering/5$ ./preprocessor
mike@system76-pc:~/C-SoftwareEngineering/5$ g++ -D_DEBUG -std=c++17 preprocessor.cpp -o preprocessor
mike@system76-pc:~/C-SoftwareEngineering/5$ ./preprocessor
Pi is:3.1415
mike@system76-pc:~/C-SoftwareEngineering/5$
```

Some General Tips on Code Writing and Debugging



List of Tips to write better software and ease debugging

- Use defensive programming techniques to strengthen your code
 - use assert and static_assert
 - Break your program into smaller pieces (modularize as necessary)
- Do write tests
- Do think a little bit before writing code
 - Explain to someone else, draw a picture, etc.
- Make very small changes to programs, then proceed to add more
- Take breaks
 - Walk away, and revisit the problem a little later when your mind is fresh

Closing Thoughts (1/2)

- **Question to audience:** Are there any weaknesses to debugging?

Closing Thoughts (2/2)

- **Question to audience:** Are there any weaknesses to debugging?
 - One thing to consider is 'code coverage' and this comes hand-in hand with testing
 - We'll only be able to use our debugging skills on portions of the code that actually executes
 - There's also some difficulty of debugging optimized builds
 - Some debuggers support this better than others

A 'general list' of tools for debugging

(Somewhat bias towards Linux)

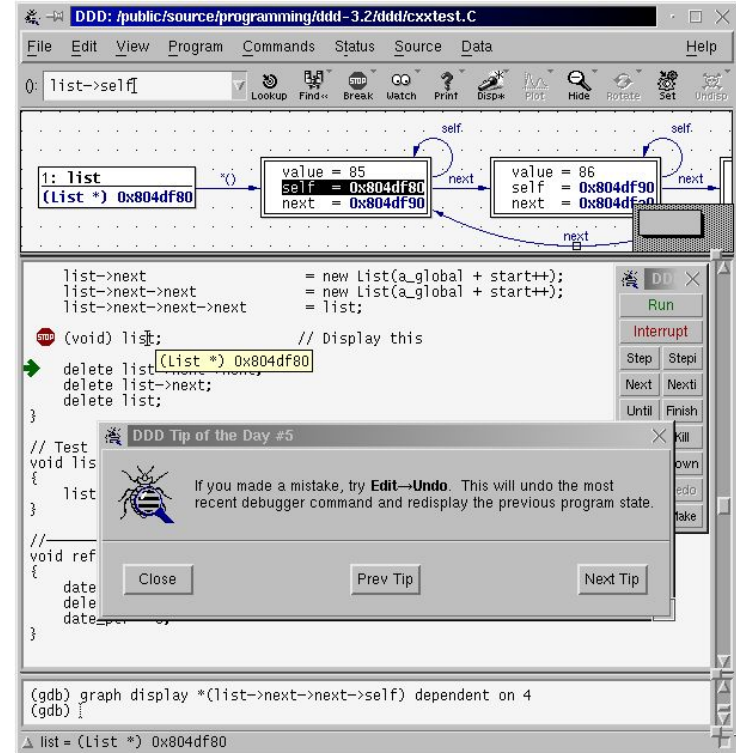
Debugging Tools List

- **Debuggers**
 - GDB
 - LLDB
- **Profilers**
 - perf
 - Intel VTune
- **Systems tools**
 - strace
 - ltrace
 - dtrace (mac)
- **Binary Analysis Tools**
 - objdump (linux)
 - otool (mac)
 - Dependency Walker (win)
- **Static Analysis Tools**
 - See 'sanitizers' for your compiler (e.g. asan, tsan)
 - cppcheck

More debugging resources - DDD (1/2)

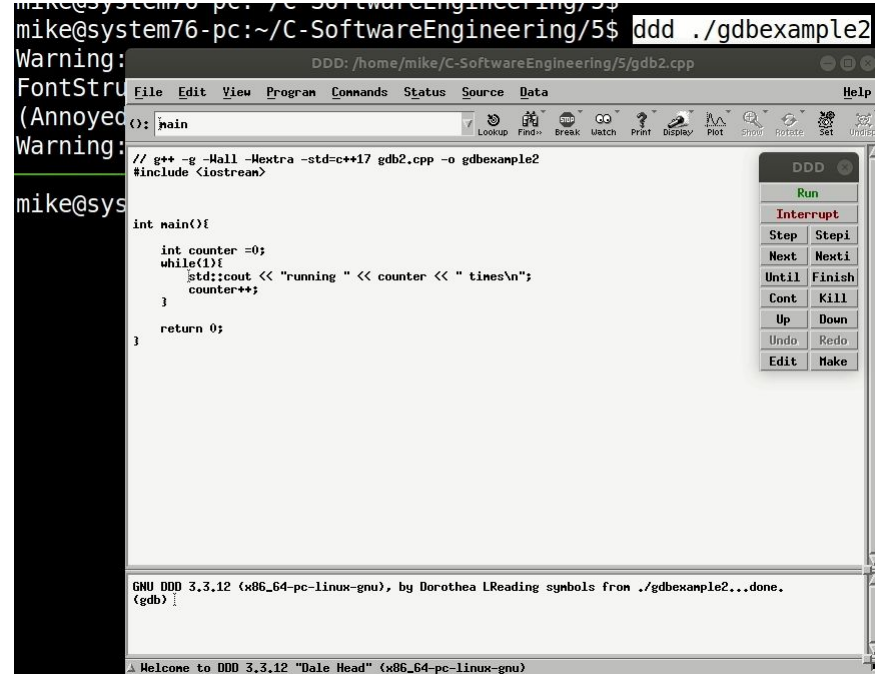
You are welcome to explore more tools and use them in this course

- A visual debuggers like (DDD) may be helpful.
 - This debugger visualizes data structures
 - <https://www.gnu.org/software/ddd/>
- Tools like source trail or other tools may additionally help you investigate and learn about your codebase.



More debugging resources - DDD (2/2)

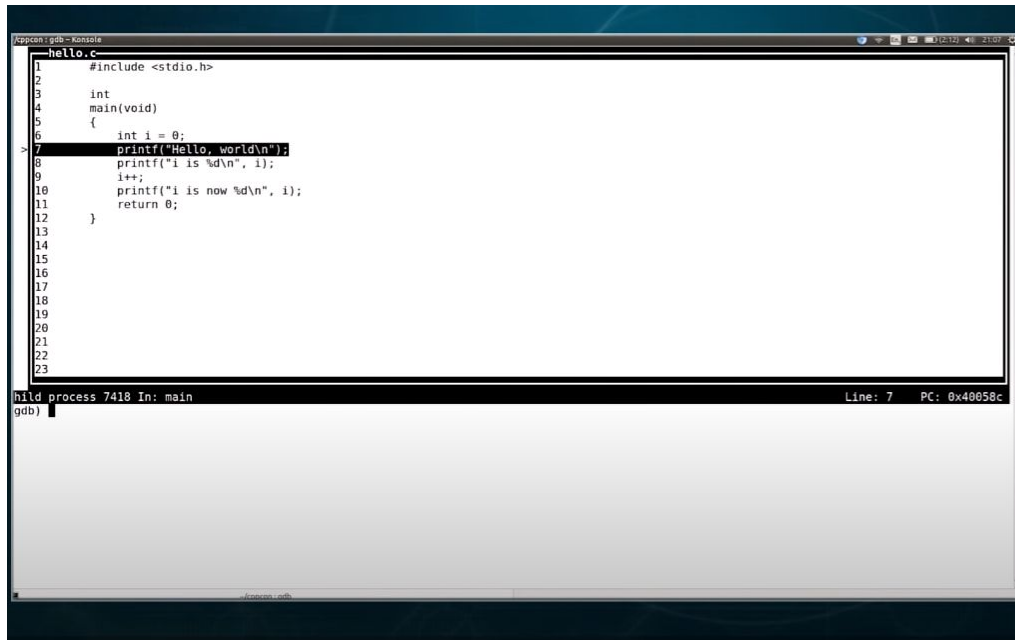
- Here's an example of DDD in practice
 - Launched with: `ddd ./gdbexample2`
- Uses the same gdb commands we learned, but also a GUI interface
 - This tool works on Linux
 - The point of me showing you this, is other IDEs you use (CLion, Visual Studio, XCode) provide nice interfaces as well.



Debugging Specific Talks

CppCon 2015: Greg Law " Give me 15 minutes & I'll change your view of GDB"

- If you have 15 minutes (which you do), watch this talk



The screenshot shows a GDB console window with the following content:

```
hello.c
1  #include <stdio.h>
2
3  int
4  main(void)
5  {
6      int i = 0;
7      printf("Hello, world\n");
8      printf("i is %d\n", i);
9      i++;
10     printf("i is now %d\n", i);
11     return 0;
12 }
13
14
15
16
17
18
19
20
21
22
23

hit process 7418 In: main
gdb) 
```

The status bar at the bottom indicates "Line: 7 PC: 0x40058c".



GREG LAW

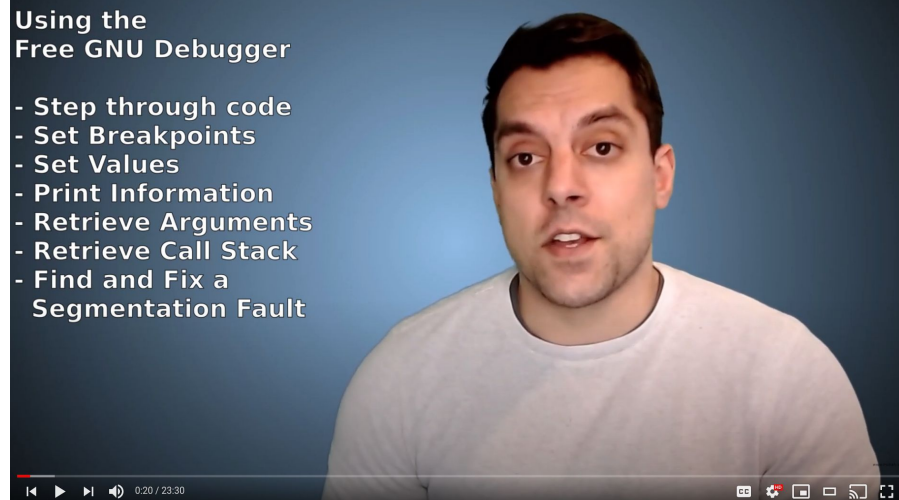
Give me fifteen
minutes and I'll
change your
view of GDB

My minimal subset of skills for students (and cheatsheet for you today)

- `gdb ./prog` to start the program
- `gdb` and the file `./prog` to reload a program after changes
- `'n'` or `'next'` to move to next line
- `'l'` or `'list'` to list source code.
- `'Ctrl+x 1'` to enter the TUI mode.
 - or `'layout src'` (`'help layout'` for more)
- `'Ctrl+x o'` to shift between windows
 - or `'focus cmd'` and `'focus src'`
- `'s'` or `'step'` to step into the source code.
- `'br'` or `'break'` to set a breakpoint followed by a line number or function name
- `'c'` or `'continue'` to continue from a breakpoint.
- `'set var=value'` to set a variable value.
- `'p'` or `'print'` followed by a variable.
 - Note: You can also dereference a variable (e.g. `print *px`) to see the dereferenced value.
- `'bt'` or `'backtrace'` to get the stack frame.
- `'f'` or `'finish'` to execute a function to completion.
- `'info args'` to get information about the incoming function arguments'

Using the Free GNU Debugger

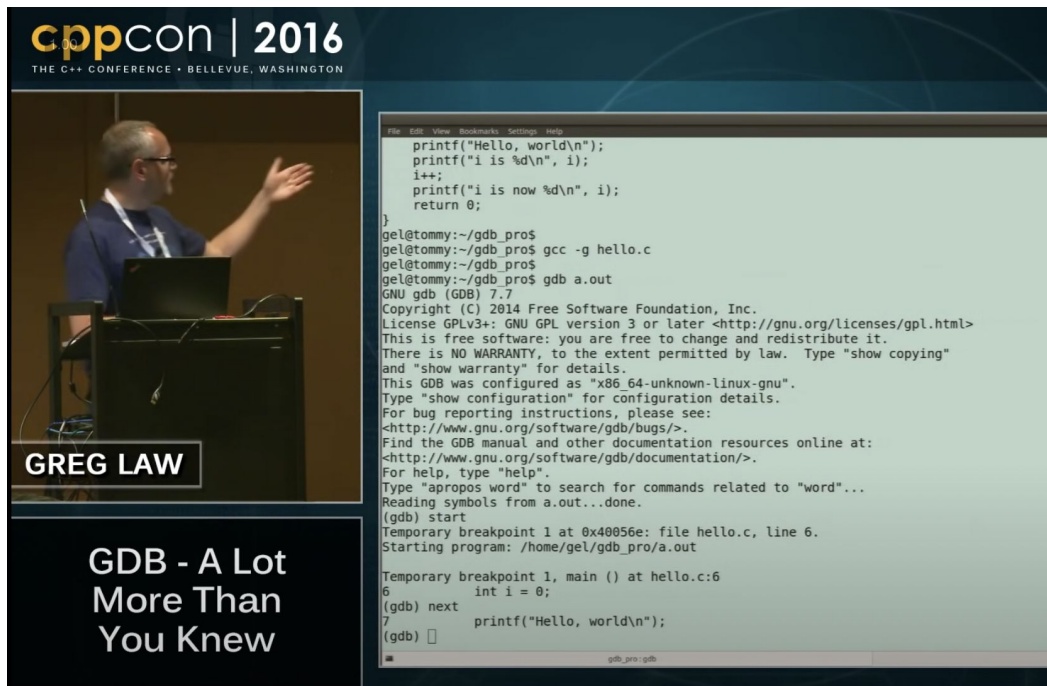
- Step through code
- Set Breakpoints
- Set Values
- Print Information
- Retrieve Arguments
- Retrieve Call Stack
- Find and Fix a Segmentation Fault



- [GDB Beginner Masterclass](#) (23 minutes on YouTube)
- Nearly all of my students see and practice this at a minimum in every applicable course I teach

CppCon 2016: Greg Law “GDB - A Lot More Than You Knew”

- Using GDB more in-depth



cppcon | 2016
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

GREG LAW

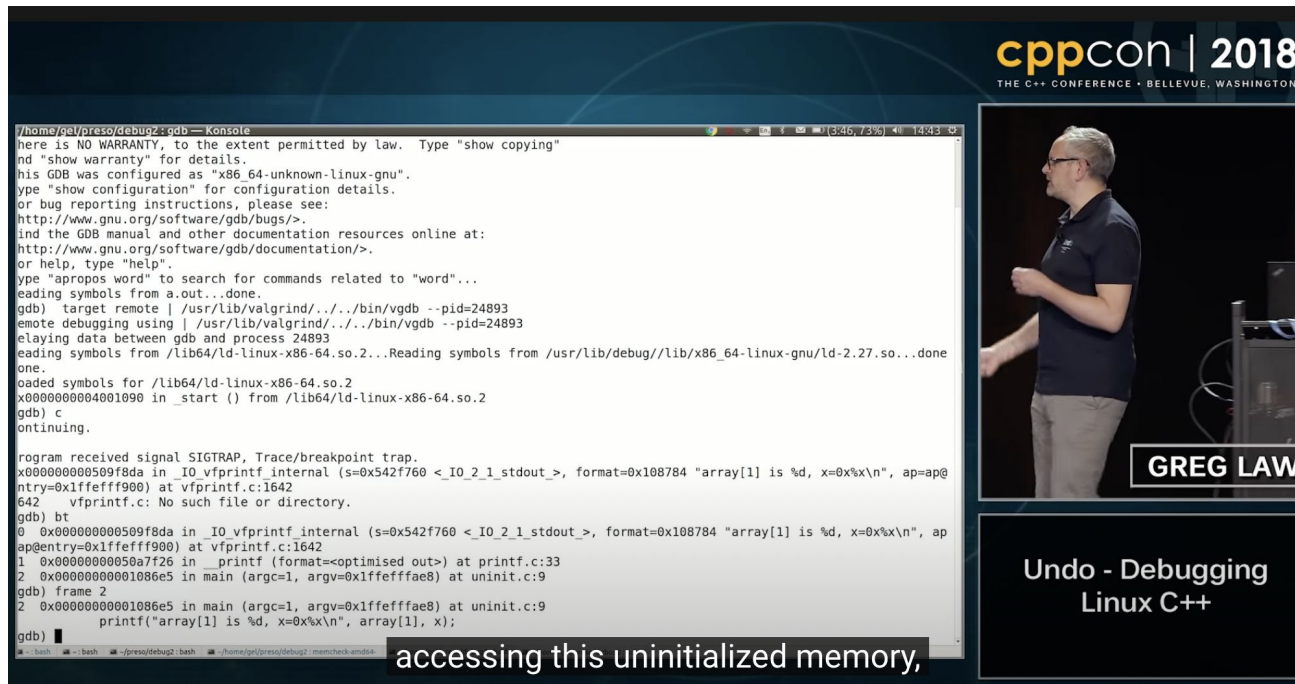
GDB - A Lot More Than You Knew

```
File Edit View Bookmarks Settings Help
printf("Hello, world\n");
printf("i is %d\n", i);
i++;
printf("i is now %d\n", i);
return 0;
}
gel@tommy:~/gdb_pro$
gel@tommy:~/gdb_pro$ gcc -g hello.c
gel@tommy:~/gdb_pro$
gel@tommy:~/gdb_pro$ gdb a.out
GNU gdb (GDB) 7.7
Copyright (C) 2014 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-unknown-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 a.out...done.
(gdb) start
Temporary breakpoint 1 at 0x40056e: file hello.c, line 6.
Starting program: /home/gel/gdb_pro/a.out

Temporary breakpoint 1, main () at hello.c:6
6      int i = 0;
(gdb) next
7      printf("Hello, world\n");
(gdb) 
```

CppCon 2018: Greg Law “Debugging Linux C++”

- GDB and other various tools (strace, asan, etc.)



The image shows a presentation slide for CppCon 2018. On the left, a GDB terminal window displays the following text:

```
/home/gel/preso/debug2-gdb - Konsole
here is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
this GDB was configured as "x86_64-unknown-linux-gnu".
type "show configuration" for configuration details.
or 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/>>.
or help, type "help".
type "apropos word" to search for commands related to "word"...
loading symbols from a.out...done.
gdb) target remote | /usr/lib/valgrind/../../bin/vgdb --pid=24893
remote debugging using | /usr/lib/valgrind/../../bin/vgdb --pid=24893
relaying data between gdb and process 24893
loading symbols from /lib64/ld-linux-x86-64.so.2...Reading symbols from /usr/lib/debug//libx86_64-linux-gnu/ld-2.27.so...done
done.
loaded symbols for /lib64/ld-linux-x86-64.so.2
0x000000004001090 in _start () from /lib64/ld-linux-x86-64.so.2
gdb) c
continuing.


rogram received signal SIGTRAP, Trace/breakpoint trap.
0x00000000509f8da in _IO_vfprintf_internal (s=0x542f760 <_IO_2_1_stdout_>, format=0x108784 "array[1] is %d, x=0x%x\n", ap=ap@
entry=0x1ffff900) at vfprintf.c:1642
1642 vfprintf.c: No such file or directory.
gdb) bt
0 0x00000000509f8da in _IO_vfprintf_internal (s=0x542f760 <_IO_2_1_stdout_>, format=0x108784 "array[1] is %d, x=0x%x\n", ap
ap@entry=0x1ffff900) at vfprintf.c:1642
1 0x0000000050a7f26 in __printf (format=<optimised out>) at printf.c:33
2 0x0000000001086e5 in main (argc=1, argv=0x1fffffae8) at uninit.c:9
gdb) frame 2
2 0x0000000001086e5 in main (argc=1, argv=0x1fffffae8) at uninit.c:9
printf("array[1] is %d, x=0x%x\n", array[1], x);
gdb) █
```

Below the terminal window, the text "accessing this uninitialized memory," is displayed.

On the right, a photo of Greg Law is shown. He is wearing a dark blue polo shirt and light-colored trousers. The text "GREG LAW" is overlaid on the photo. Below the photo, the text "Undo - Debugging Linux C++" is displayed.

Cool New Stuff in Gdb 9 and Gdb 10 - Greg Law - CppCon 2021

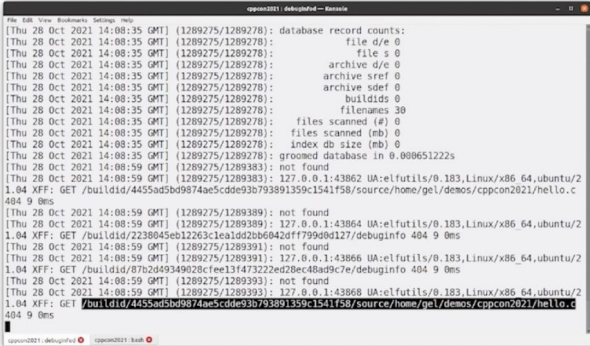
- No code, but ways to better approach the task of debugging



Greg Law

Cppcon
The C++ Conference

Cool New Stuff in Gdb 9, Gdb 10 and Gdb 11



```
cppcon2021: debuginfod - Remote
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289278): database record counts:
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289278): file d/e 0
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289278): file s 0
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289278): archive d/e 0
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289278): archive sref 0
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289278): archive sdef 0
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289278): buildids 0
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289278): filenames 30
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289278): files scanned (#) 0
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289278): files scanned (mb) 0
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289278): index db size (mb) 0
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289278): groomed database in 0.000651222s
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289383): not found
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289383): 127.0.0.1:43862 UA:elfutils/0.183.Linux/x86_64.ubuntu/2
1.04 XFF: GET /buildid/4455ad5bd9874ae5cde93b793891359c1541f58/source/home/gel/demos/cppcon2021/hello.c
404 9 0ms
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289389): not found
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289389): 127.0.0.1:43864 UA:elfutils/0.183.Linux/x86_64.ubuntu/2
1.04 XFF: GET /buildid/2238045eb1263c1eald62b6042df7799d0d127/debuginfo 404 9 0ms
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289391): not found
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289391): 127.0.0.1:43866 UA:elfutils/0.183.Linux/x86_64.ubuntu/2
1.04 XFF: GET /buildid/87b2d49349028cfee13f47322ed28ec48ad9c7e/debuginfo 404 9 0ms
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289393): not found
[Thu 28 Oct 2021 14:08:35 GMT] (1289275/1289393): 127.0.0.1:43868 UA:elfutils/0.183.Linux/x86_64.ubuntu/2
1.04 XFF: GET /buildid/4455ad5bd9874ae5cde93b793891359c1541f58/source/home/gel/demos/cppcon2021/hello.c
404 9 0ms
```


HTTP

debuginfod

DEBUGINFOD_URLS=localhost:8002 gdb a.out

Back To Basics: Debugging Techniques - Bob Steagall - CppCon 2021

- No code, but ways to better approach the task of debugging



Bob Steagall

Back to Basics:
Debugging Techniques

Cppcon 2021 | The C++ Conference | October 24-29

The Cost of Software Failures

- January 2018, Tricentis' *Software Fail Watch* documents 606 software failures in CY 2017
 - 3.6 billion people affected
 - \$1.7 trillion lost revenue
 - Software failures resulted in 268 years of downtime
 - The number of reported failures was 10 percent higher in 2017 than in 2016
 - Retail and consumer technology industries experienced the most software failures of any industry analyzed

CppCon 2021 - Back to Basics: Debugging Techniques | Copyright © 2021 Bob Steagall

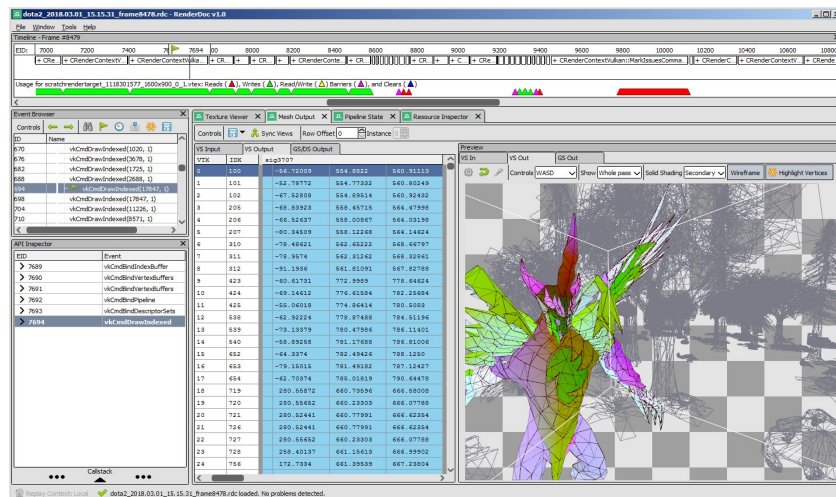
Outside the scope of this lecture

- GPU Debugging
 - NVidia Nsight
 - One example -- most vendors (i.e. AMD, Intel, etc. have their own debuggers as well)
 - Renderdoc GPU debugger
 - <https://renderdoc.org/>
- More on Time Travel debugging
 - <https://rr-project.org/> -- time travel debugger (next step after gdb!)
 - <https://undo.io/solutions/products/udb/> -- UndoDB debuggers
- Greg Laws resources on more debugging
 - <https://undo.io/resources/gdb-watchpoint/>



NVIDIA Nsight Systems

NVIDIA® Nsight™ Systems is a system-wide performance analysis tool designed to visualize application's algorithm, help you select the largest opportunities to optimize, and tune to scale efficiently across any quantity of CPUs and GPUs in your computer; from laptops to DGX servers.



Thank you!

+ 22

Back To Basics Debugging

Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)
YouTube: [www.youtube.com/c/MikeShah](#)



IAH



September 12th-16th

