

Back to Basics: Debugging Techniques

Bob Steagall
CppCon 2021



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
- May 2020, Undo and a Judge Business School MBA project found that
 - Reproducing and fixing a failure takes about 13 hours on average
 - About **26%** of developer time is spent reproducing and fixing failures
 - **620M** developer hours/year -- **\$61B** salary -- **\$1.2T** enterprise value lost for shareholders

The Cost of Software Failures

- Radiation overdoses from Therac-25 cause death
 - Race condition
- Ariane 5 explodes at launch (\$370m)
 - double to int16 conversion
- Mars climate orbiter burns up in space (\$235m)
 - Imperial to metric conversion
- Knight Capital Loses \$460m in 45 minutes
 - Dead code, test flag accidentally left enabled
- Boeing 737 MAX MCAS system
 - System component design flaws

- What are bugs?
- What is debugging?
- Challenges when debugging
- A simple process for debugging
- Some recommendations

Warning! Warning! WARNING!



Opinions lie ahead!

What are defects?

- The most common view
 - A software defect (bug) is an *error* in a computer program causes it to produce incorrect or unexpected results, or exhibit unintended behavior
- Another way of thinking about it
 - Each software system is subject to a set of **requirements** that describe its operating environment, inputs, usage, interactions, outputs, and expected behavior
 - These requirements are not always explicitly stated, but are always present
 - A software defect is then a *non-conformity to requirements*
 - Some subset of the system's requirements that are being violated
 - This viewpoint is very common in regulated industries

What is Debugging?

- Wikipedia says:

"... **debugging** is the process of finding and resolving *bugs* (defects or problems that prevent correct operation) within computer programs, software, or systems."

- Assumptions

- We know what *correct operation* is and we know what *incorrect operation* is
- We have the ability to observe the programs and/or their output
- We have the ability to change the underlying source code and other program data
- We have the ability to build and test the updated programs

Some Terminology

- A *non-conformity* is a failure to meet one or more requirements
- A *defect* (*bug* / *error* / *problem*) is incorrect program data (code, input, settings, dependencies, ...) that causes a *non-conformity*
- A *symptom* is observable evidence of a *defect*
- A *deterministic defect* is a *defect* that does not change its *symptoms* under a well-defined set of conditions
- In contrast, a *non-deterministic defect* is a *defect* that changes its *symptoms* from run-to-run under a well-defined set of conditions

Some Terminology

- A *context* is the totality of the environment in which a program that exhibits symptoms is running
- A *problem report* describes one or more *symptoms* in some *context*
- An *analogous context* is one that replicates enough of the original context sufficiently to reproduce a set of *symptoms*
- The *lab* is a setting in which you have total control over the context
- The *field* is a setting where you have minimal or no control over the context

Some Terminology

- A *problem report* describes one or more *symptoms* in some *context*
- Each *symptom* is caused by one or more *defects* (*errors, problems*)
- Evidence of a *defect* is made observable by one or more *symptoms*



Debugging Challenges

- Problem reports may be "unhelpful"
 - Misleading / inadequate / incorrect description of symptoms
 - Lack of knowledge about the product – version / configuration / platform / etc.
- Problem reports may not indicate actual problems
 - Unexpected behavior is not always a defect
- Collecting program state data may be difficult
 - Log / settings / crash data could be incomplete, inconclusive, unavailable, non-existent
- Symptoms may not indicate the cause
 - The cause and the effect may be distant, in space or time

Debugging Challenges

- Defects and symptoms may be correlated
 - Sometimes symptoms change as repair progresses
- Fixing one defect may introduce new defects
 - Indicative of quick-fixes and/or messy design
- **Symptoms may be difficult to reproduce**
 - Debugging the field is not always possible
 - Constructing an analogous context in the lab is not always feasible
 - Program output is not always available
 - Symptoms from non-deterministic problems can be especially challenging

The Debugging Process – In Theory

- Characterize and reproduce
 - Determine the surrounding context and observe incorrect behavior
- Locate
 - Find the lines of code responsible for the defect
- Classify
 - Decide what kind of defect you have
- Understand
 - Determine the root cause of the defect and its relationship to the whole
- Repair
 - Resolve the defect without breaking anything else

The Debugging Process – In Theory

- We tend to think of debugging as a simple linear process

```
Product MyJob::Debug(Product const& curr, Problem const& issue)
{
    CharacterizeAndReproduceProblem(curr, issue);

    LocateProblem(curr);

    ClassifyProblem(curr);

    UnderstandProblem(curr);

    Product next = RepairProblem(curr);

    return next;
}
```

- The process appears to make sense, however...

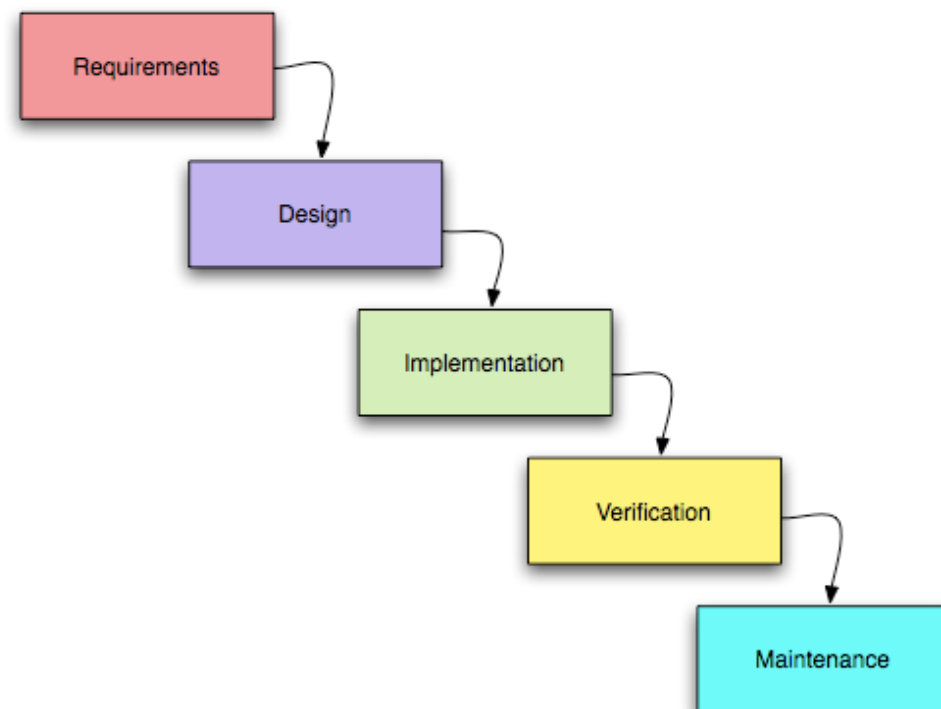
"... in theory there is no difference between theory and practice,
while in practice there is"

- *Portfolio: Theory and Practice* by Benjamin Brewster, 1882 February,
The Yale Literary Magazine
- Famously mis-attributed to Yogi Berra, Albert Einstein, Richard Feynman,
Walter Savitch, and others

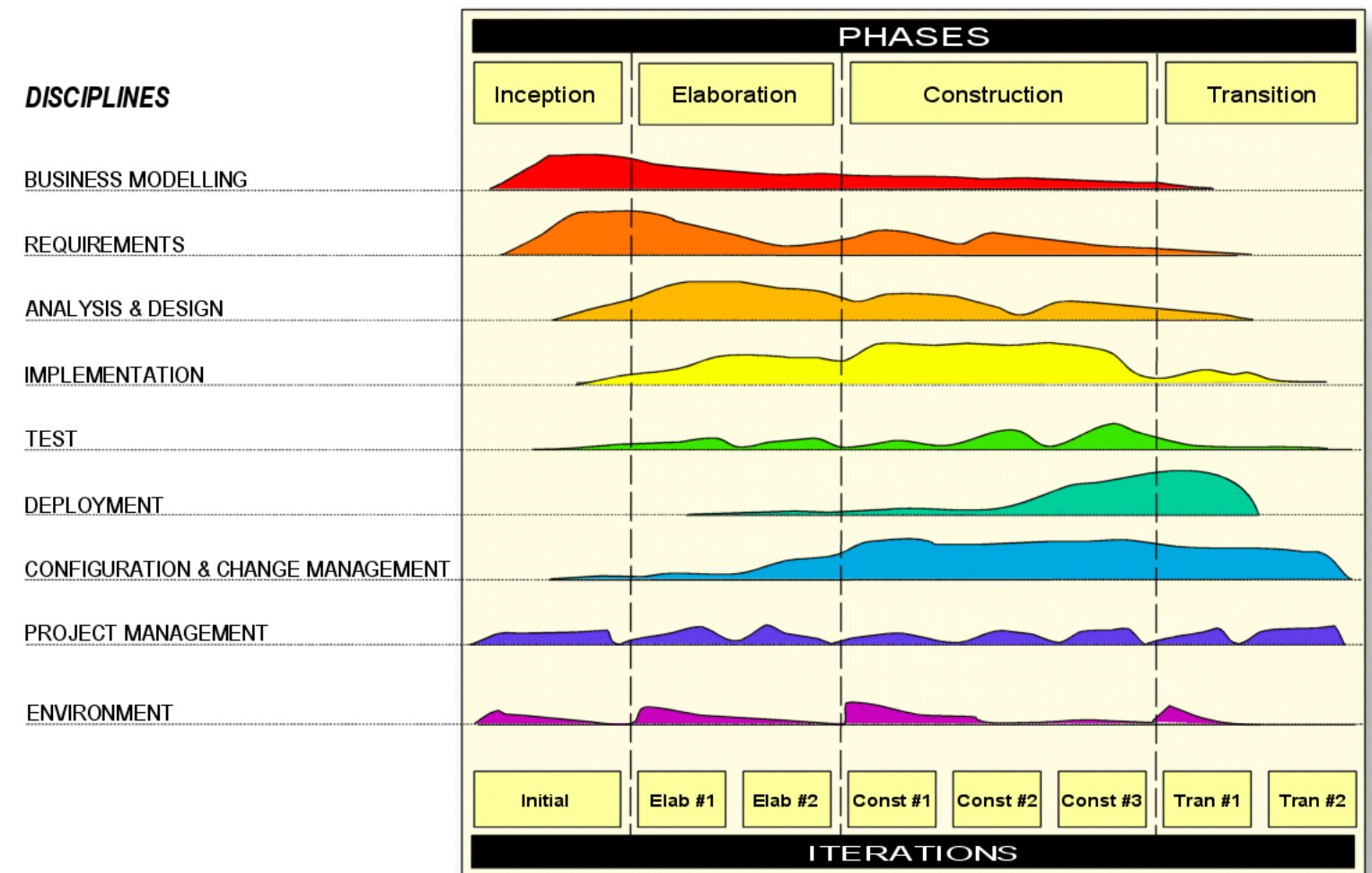
The Debugging Process – In Theory – And Practice

- There's an analogy here with software engineering processes...

Simple cases - waterfall



Complex cases – iterative and incremental



The Debugging Process – In Practice

```
bool MyJob::Debug(Product const& curr, Problem const& issue)
{
    ReviewProblemReport(curr, issue);
    CharacterizeAndReproduceProblem(curr);
    auto next = Clone(curr);

    while (ReproduceProblem(next) && ResourcesAvailableToRepair(next))
    {
        auto insight = async(launch::async, &MyJob::UnderstandProblem, this, next);
        auto location = async(launch::async, &MyJob::LocateProblem, this, next);
        auto category = async(launch::async, &MyJob::ClassifyProblem, this, next);
        WaitFor(insight, location, category);

        next = AttemptToRepair(next);
    }

    return (ProblemFixed()) ? Deliver(next), true
                           : PossiblyUpdateResume(), false;
}
```

The Debugging Process – In Practice

```
bool MyJob::Debug(Product const& curr, Problem const& issue)
{
    ReviewProblemReport(curr, issue);
    CharacterizeAndReproduceProblem(curr);
    auto next = Clone(curr);

    while (ReproduceProblem(next) && ResourcesAvailableToRepair(next))
    {
        auto insight = async(launch::async, &MyJob::UnderstandProblem, this, next);
        auto location = async(launch::async, &MyJob::LocateProblem, this, next);
        auto category = async(launch::async, &MyJob::ClassifyProblem, this, next);
        WaitFor(insight, location, category);

        next = AttemptToRepair(next);
    }

    return (ProblemFixed()) ? Deliver(next), true
                           : PossiblyUpdateResume(), false;
}
```

Characterizing and Reproducing a Problem

- **Characterizing**

- Determining the context in which symptoms were observed
- Version number, platform, resources allocated, external interfaces, configuration data, etc.
- Information that allows you to instantiate an analogous context

- **Reproducing**

- Instantiating an analogous context, in the lab, or in the field
- Running enough of the program (or the system) to observe the reported symptoms
- Developing new / updating existing test assets to demonstrate the failure
- Pro tip: Make sure you're looking at the correct source code!

- **Characterizing and reproducing a problem is vital to the debugging process**

The Debugging Process – In Practice

```
bool MyJob::Debug(Product const& curr, Problem const& issue)
{
    ReviewProblemReport(curr, issue);
    CharacterizeAndReproduceProblem(curr);
    auto next = Clone(curr);

    while (ReproduceProblem(next) && ResourcesAvailableToRepair(next))
    {
        auto insight = async(launch::async, &MyJob::UnderstandProblem, this, next);
        auto location = async(launch::async, &MyJob::LocateProblem, this, next);
        auto category = async(launch::async, &MyJob::ClassifyProblem, this, next);
        WaitFor(insight, location, category);

        next = AttemptToRepair(next);
    }

    return (ProblemFixed()) ? Deliver(next), true
                           : PossiblyUpdateResume(), false;
}
```

Understand the Problem

- **Understanding**

- Gaining enough knowledge about a problem and the surrounding code, that you believe you can make changes to carry out a repair
- At a minimum, you should have
 - Located the incorrect lines of code
 - Determined why the code is incorrect – what is the root cause?
 - Determined a proposed classification
 - Formulated a set of proposed changes
 - Determined how your proposed changes would affect the runtime state
 - Determined if your proposed changes would correct the problem

Understand the Problem

- Inspect and verify the associated test assets
 - The test case(s) or harness(es) may be broken
 - If appropriate, test data should demonstrate correct and incorrect behavior
- The defect may not be where you expect it
 - Keep an open mind and be ready to question all parts of the program
- Ask yourself where the defect is *not*
 - Sometimes trying to prove the absence of a defect reveals the defect
- Explain to yourself (or someone else) why there is a defect, and why your proposed fix will resolve the defect
 - A local guru – or CppCon bobblehead – could be helpful

Locate the Problem

- Employ good development practices at the outset
 - Practice iterative, incremental, bottom-up development
 - Add functionality in small sections of code
 - Create test assets for each new increment of functionality
 - Verify that new code doesn't cause previous test cases to fail
 - Verify that new code passes its own test cases
 - Practice defensive programming
- Comments
 - It's much easier to find defects in modular, well-designed code with well-written and extensive test assets
 - Preferably the whole product does this, at a minimum your fixes should

Locate the Problem

- Use trace logging
 - Generating output describing the program state during execution
 - In simpler cases, instrument code with print statements
 - In more complex systems, take advantage of existing logging facilities
- Comments
 - Great way to stay "on the path" when developing new code
 - An easy first step in narrowing down a problem's scope
 - Adds runtime overhead, which can hinder the search for non-deterministic problems

Locate the Problem

- Use debugging and analysis tools
 - Your C++ compiler (*warnings*)
 - Static code analysis tools (*coverity, cppcheck, ...*)
 - Interactive debuggers (*gdb, lldb, msvc, udb, ...*)
 - Time-travel debuggers (*gdb, rr, liverecorder, udb, ...*)
 - Sanitizers (*asan, tsan, ubsan, ...*)
 - Dynamic program analyzers (*valgrind, callgrind, helgrind, ...*)
 - Call tracers and domain-specific diagnostic tools (*strace, wireshark, SQL analyzers...*)
- Comments
 - Very powerful tools for deterministic problems
 - Not always useful for non-deterministic problems, especially if runtime overhead is added

Locate the Problem

- Enable and/or add assertions
 - An *assertion* is a facility that evaluates a Boolean predicate at runtime, and causes a serious error (program termination or C++ exception) if the predicate is **false**
 - Verify pre-conditions/post-conditions before/after calling a function
 - Verify class invariants (e.g., in a **self_test()** member function)
 - Verify expected program state
- Comments
 - Usually has little effect on execution speed
 - Good tool for both deterministic and non-deterministic defects
 - Adds some runtime overhead and code complexity

Locate the Problem

- Use backtracking
 - Start where you think the problem occurred and step backward through the code
 - Try to understand the program state at each backward step
- Comments
 - Good for very simple programs and/or small search with having deterministic problems
 - Easy to do quickly and form a first guess
 - Less effective in complex programs or large search areas
 - Usually ineffective for non-deterministic problems

Locate the Problem

- Divide and conquer (binary search)
 - Pick section of code to examine
 - Place an assertion, or set a breakpoint, halfway through the section
 - If the assertion fires, or the breakpoint is reached with an invalid program state, then the problem is in the first half; otherwise, it's in the second half
 - Repeat this halving process until you get to a section that reveals the defect
- Comments
 - Trace logging can be used in place of assertions or breakpoints
 - For deterministic problems with a large search area, this is a very powerful technique
 - Not always effective for non-deterministic problems

Locate the Problem

- Problem simplification
 - Gradually and strategically remove (comment out) sections of irrelevant code
 - Can be combined with divide-and-conquer
 - Can be applied to input data as well

- Comments
 - When combined with divide-and-conquer, this is very useful for debugging crashes of release builds as well as non-deterministic problems
 - Work backwards from the end of the section, removing the second half each time

Locate the Problem

- Make the problem worse
 - Vary the context in order to evoke the symptoms (force failures) more frequently
- Comments
 - Figuring out how to make the problem worse is often a helpful first step in finding and understanding the problem
 - Very useful for non-deterministic problems, especially when the symptoms are infrequent

Locate the Problem

- Scientific method
 - Form a hypothesis consistent with observations
 - Implement test(s) to refute the hypothesis
 - If refuted, form a new hypothesis and implement new test(s) to refute it
 - Repeat until your hypothesis cannot be refuted
 - Requires active thinking about the problem
- Comments
 - Effective for all problems
 - Forces you to really understand the code and how program state evolves
 - Is compatible with the other methods of location
 - Can be very time-consuming for unfamiliar code bases

Locate the Problem

- Deterministic problems
 - Review the logs
 - Add assertions where needed to verify invariants
 - If possible, use an interactive debugger, using divide-and-conquer
 - Otherwise, add assertions, using divide-and-conquer
- Non-deterministic problems
 - Review the logs
 - Create a debug build and see if it also exhibits the same symptoms
 - Add assertions where needed to verify invariants
 - Add assertions, and/or comment out code, using divide-and-conquer
 - **Try to make the problem worse**
 - If needed, try low-overhead debugging tools

Classifying a Problem

- **Classifying**
 - Determining a defect's *category*
 - Can be useful in formulating a repair strategy
 - Important information in subsequent reviews when considering preventive actions
- **Syntax errors**
 - Invalid source code – violates the language's grammar
 - In C++, the compiler catches these and issues **errors** (mostly)
 - More of an issue in multi-platform products
- **Syntax warnings**
 - Basic semantic errors – syntactically valid, but *questionable* code
 - In C++, the compiler issues **warnings**

Classifying a Problem

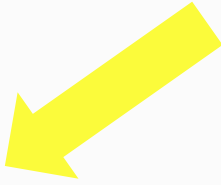
- Simple source code errors
 - IOW, syntactically correct typos
 - Permit successful builds with latent defects

```
class line_buffer {...};

bool read_line(line_buffer& b);
void parse_line(line_buffer const& b);

void parse_file()
{
    line_buffer buf;

    while (read_line(buf));
    {
        parse_line(buf);
    }
}
```



Classifying a Problem

- Implementation errors
 - High-level algorithms / data structures / workflows are correct, but
 - Lower-level data structures are used incorrectly
 - Broken invariants, pre-/post-condition violations
- Logic errors
 - High-level algorithms / workflows are logically flawed
 - Mostly correct operation, but fails on corner cases
 - Indicative of design flaws
- Configuration (build) errors
 - Incorrect / invalid binary components are included in a build
 - Result in successful builds that appear to work

The Debugging Process – In Practice

```
bool MyJob::Debug(Product const& curr, Problem const& issue)
{
    ReviewProblemReport(curr, issue);
    CharacterizeAndReproduceProblem(curr);
    auto next = Clone(curr);

    while (ReproduceProblem(next) && ResourcesAvailableToRepair(next))
    {
        auto insight = async(launch::async, &MyJob::UnderstandProblem, this, next);
        auto location = async(launch::async, &MyJob::LocateProblem, this, next);
        auto category = async(launch::async, &MyJob::ClassifyProblem, this, next);
        WaitFor(insight, location, category);

        next = AttemptToRepair(next);
    }

    return (ProblemFixed()) ? Deliver(next), true
                           : PossiblyUpdateResume(), false;
}
```

Repairing the Problem

- **Repairing**
 - Implementing the *appropriate* set of source code changes that are *necessary* and *sufficient* to resolve the defect
 - Demonstrate resolution by passing tests
- Try to minimize changes to the system – keep changes small and localized
 - Static changes – source code
 - Dynamic changes – program state at runtime
- Verify your repairs against your test assets
 - All new/updated tests should pass
 - All other tests that previously passed must continue to pass

The Debugging Process – In Practice

```
bool MyJob::Debug(Product const& curr, Problem const& issue)
{
    ReviewProblemReport(curr, issue);
    CharacterizeAndReproduceProblem(curr);
    auto next = Clone(curr);

    while (ReproduceProblem(next) && ResourcesAvailableToRepair(next))
    {
        auto insight = async(launch::async, &MyJob::UnderstandProblem, this, next);
        auto location = async(launch::async, &MyJob::LocateProblem, this, next);
        auto category = async(launch::async, &MyJob::ClassifyProblem, this, next);
        WaitFor(insight, location, category);

        next = AttemptToRepair(next);
    }

    return (ProblemFixed()) ? Deliver(next), true
                           : PossiblyUpdateResume(), false;
}
```

Delivering the Fix

- **Delivery**
 - Incorporating changes that repair a problem into production code
- Practice good version control
 - Don't include fixes for more than one problem in one commit
 - Don't include extraneous changes (e.g., new features) in fix commits
 - Include new/updated test assets in the fix commits
 - Write commit comments that are clear and concise
- Verify your tests, again
 - Double-check that all new/updated tests pass
 - Double-check that all other tests that previously passed continue to pass

Delivering the Fix

- Create documentation for posterity
 - How the defect was noticed
 - The conditions under which the defect occurred – the context
 - Steps necessary to reproduce the defect – the analogous context
 - Techniques and tools used to localize the defect
 - The defect's category
 - The underlying root cause of the defect
 - Latent defects precluded by fixing this defect
 - Possible latent defects left unaddressed
 - Mistake made and recommendations for preventive actions

- Conduct required reviews (if reviews are part of your SWE process)

Summary / Recommendations

- Practice **defensive programming**
 - Assume the worst case could happen at any time – it usually will
- Employ an appropriate **iterative and incremental development process**
 - Decide what needs to be achieved
 - Formulate a plan for the achievement
 - Understand the invariants, requirements, and context, then design the solution
 - Implement the solution in small, discrete, testable chunks
 - Write code to verify invariants, pre-conditions, post-conditions, and self-test complex components
 - Consider employing the principles of test-driven design
 - Implement test assets in parallel with the solution
 - Employ good configuration management practices everywhere

Summary / Recommendations

- Reproduce the problem!
 - If not in the lab, try to reproduce it in the field
- Learn to use the tools at your disposal
 - Enable C++ compiler warnings
 - Static code analysis tools (*coverity*, *cppcheck*, ...)
 - Interactive debuggers (*gdb*, *lldb*, *msvc*, *udb*, ...)
 - Time-travel debuggers (*gdb*, *rr*, *liverecorder*, *udb*, ...)
 - Sanitizers (*asan*, *tsan*, *ubsan*, ...)
 - Dynamic program analyzers (*valgrind*, *callgrind*, *helgrind*, ...)
 - Call tracers and domain-specific diagnostic tools (*strace*, *wireshark*, *SQL analyzers*...)

Summary / Recommendations

- Try to leave the code better, if only a little
 - Employ defensive programming and a good process, even if only for your repairs
- Don't try to refactor
 - Repair only what needs to be repaired – don't mix repair and refactoring
- Thoroughly document your changes and their justifications
 - Leave breadcrumbs for the next person
- Test your changes – make sure they work!
 - If necessary, create new tests and/or update existing test assets
 - Run existing tests and verify no regressions have been introduced
 - Test in the lab, and if possible, test in the field

Questions? Thank You for Attending!

Talk: github.com/BobSteagall/CppCon2021

Blog: bobsteagall.com