A presentation has three parts. An intro, a body, and a conclusion. In the intro you tell them what you're going to tell them.

Today I'll tell you how to construct a presentation. Presentations have three parts. An intro, a body, and a conclusion. In the intro you tell them what you're going to tell them.

I feel like I've heard this before.

I'm afraid we're going to hear it again.
Introduction

- Who is this guy? What does he do? Why is he here?
- Let's play Spot the Defect!
- How can we automate defect spotting?
- Improving the Joel Test
Who is this guy?

• Compiler developer / language designer at Microsoft from 1996 through 2012
  • Visual Basic, VBScript, JScript, VS Tools for Office, C# / Roslyn
• C# analysis architect at Coverity since January 2013
  • I will use “we” inconsistently
• I have no formal background in static analysis
  • I take an engineering rather than academic approach
About Coverity

• Founded in 2003
  • Spun off of a Stanford Ph.D. research project
  • We do static analysis of C/C++, C# and Java codebases looking for quality and security defects
• \( \sim 3 \times 10^2 \) employees
• \( \sim 1.2 \times 10^3 \) customers
  • Consumer electronics, telecommunications, software, aerospace, medical devices...
• \( \sim 5 \times 10^9 \) lines of code regularly scanned
• Development offices in San Francisco, Calgary and Seattle
• Growing quickly
• Purchased by Synopsys about five minutes ago
The middle part of a presentation is called the body. This is where you tell the audience what you're telling them.

After the intro, where you told the listener what you're going to tell them, comes the body, where you tell them what you're telling them.

Couldn't you just stop at the introduction?

Then I'd be introducing nothing, and what kind of introduction is that?

In this case, an accurate one.
static int M(string s, int x, int y)
{
    int z = 0;
    if (s != null && x < y || s.Length > 0)
        z = x + y - s.Length;
    return z;
}
Spot the defect, round one (C#)

```csharp
static int M(string s, int x, int y)
{
    int z = 0;
    if (s != null && x < y || s.Length > 0)
        z = x + y;
    return z;
}
```

&& is of higher precedence than ||.

If the null check results in false then the right side of the || executes, and asks for the length of a null string, which throws.

The || could be parenthesized, but frankly this code looks like it ought to be rewritten to make the intention more clear in the first place.

Coverity’s static analyzer determines that there is a control flow path in which a known-to-be-null value of reference type is dereferenced.
Life is terrible

- C was designed for a world of slow machines with tiny memory and was a massive improvement over assembly.
  - We don't live in that world anymore, but we still use C.
- Even modern type-safe, memory-safe languages like C# and Java permit the developer to write horrid bugs.
  - No one has made a pragmatic language yet that prevents mistakes.
- Cost of fixing a defect rises exponentially:
  - Fixed while typing code in: ~$1
  - Fixed after developer testing: ~$10
  - Fixed after unit testing: ~$100
  - Fixed after integration testing: ~$1000
  - Fixed after shipping: arbitrarily expensive
- We can use software to mitigate the disasters caused by software
Enter static analysis

• Static analysis is any analysis of a program based solely on its source code or object code
  • dynamic analysis examines code as it is executing
• Compilers are static analyzers
  • Compiler errors tell you when your program is illegal, not buggy.
  • Compiler warnings tell you about potential bugs...
  • ... but are typically a fast, shallow analysis
• Many static analyzers exist for many reasons
  • Bug finding
  • Refactoring
  • Documentation
  • Software metrics
  • And so on
Jargon

- True positive: a report of a genuine flaw
- True negative: a non-report of a non-flaw
- False positive: a report of a non-flaw
- False negative: a non-report of a flaw
- Sound analysis: no false positives or negatives...
  ... is impossible
- Our goal is not to find all defects or even most defects
- "Thank goodness we found that before our customers did!"
- How do we do this in practice?
First, determine what code to analyze

- **Bad**: make the customer say what they want to analyze
- **Good**: figure it out by watching a build
- You run a clean build of whatever you want analyzed using the build capture tool to launch the build
- The tool registers itself as a debugger and/or profiler of the build process
- Invocations of compilers are detected and logged to a temporary directory
- Now we know everything about the build.
- Capture of centralized nightly build is common, but not the only possible workflow.
Second, find the defects

- Run all the source code through our own C / C++ / C# / Java compiler front end, as appropriate
  - Abstract syntax trees and source code persisted into a temporary directory

```c
while (b != 0)
    if (a > b)
        a -= b;
    else
        b -= a;
```
Compute call graph, topo-sort it
Summarize the callees

- Now that we have an acyclic interprocedural call graph we traverse the graph from leaves (callees) to roots (callers)
- As we traverse the graph each method is summarized.
  - "M(string x) dereferences its argument"
  - "OpenFile() returns a newly-allocated Stream object"
  - "Foo() uses value equality on an object of type Bar five times"
  - And so on.
- Method calls can be treated as though they had the actions described in their summaries
- Now we have enough information to find defects
Analyze each method body

• "Checkers" are FSMs whose input is a sequence of AST nodes, either in syntactic order or path order.

• Always run cheap path-insensitive analysis first
  • There can be thousands of paths through a method
  • Anything that makes analysis cheaper is goodness

• If any potential defect is found the method is analyzed again, this time with intra-procedural path-sensitivity

• **False-path-pruning** algorithms ensure defects found only on impossible paths are suppressed.

• The vast majority of lines of code are not defective
  • One defect per KLOC is typical
False path pruning

- Defects found on "impossible" paths are likely false positives (and make us look dumb)
- We use many techniques to determine if a defect is on a false path.
  - Integer interval inequality
  - Nullity tracking
  - SAT / SMT solving
  - Many secret proprietary heuristic techniques
  - Coming soon: collection invariants
Third, present the defects

• Finding defects is hard
• Presenting defects so that the developer understands them well enough to fix them is even harder
  • It is easy for skeptical, overworked, defensive developers to assume a convoluted series of events that leads to a defect is impossible
• But before we get into that...
bool linkToDatabase =
    (attr != null && attr.i_database == null) ?
        true : !isMigrating;
if (linkToDatabase)
{
    Db4oDatabase db = container.Identity();
    if (db == null)
    {
        attr = null;
    }
    else if (attr.i_database == null)
    {

Forward-Null: ~800 / MLOC

1. **var_compare_op**: Comparing `attr` to null implies that `attr` might be null.
   
   ```c
   bool linkToDatabase = (attr != null && attr.i_database == null) ? true : !isMigrating;
   ```

2. **Condition linkToDatabase, taking true branch**
   ```c
   if (linkToDatabase)
   {
     Db4oDatabase db = ((IInternalObjectContainer)container).Identity();
   }
   ```

3. **Condition db == null, taking false branch**
   ```c
   if (db == null)
   {
     attr = null;
   }
   else
   {
   }
   ```

**CID 14146 (#1 of 1): Dereference after null check (FORWARD_NULL)**

4. **null_field_access**: Accessing field of null object `attr`.
   ```c
   if (attr.i_database == null)
   {
     attr.i_database = db;
     if (container is LocalObjectContainer)
     {
       attr.i_uuid = container.GenerateTimeStamId();
       doAddIndexEntry = true;
     }
   }
   ```

[Coverity logo]
Improving the Joel Test

• You recently learned about my buddy Joel's famous 12 point test: do you have source control, easy builds, daily builds, bug tracking, aggressive bug fixing, schedules, specifications, quiet offices, software tools, testers, technical interviews, ad hoc usability testing?

• That "the best tools money can buy" item is a little vague.

• Source control and bug tracking are tools but so important that they are called out as individual items.

• I want:

13. Do you use static analysis tools to find bugs during the development process?
private int InternalCopyID(
    bool flipNegative, bool lenient, int id)
{
    if (flipNegative && id < 0)
    {
        id = -id;
    }
    int mapped = _mapping.MappedID(id, lenient);
    if (flipNegative && id < 0)
    {
        mapped = -mapped;
    }
    return mapped;
}
Dead code -- ~60 / MLOC

cond_at_most: Condition id < 0, taking true branch. Now the value of id is at most -1.

cond_at_least: Condition id < 0, taking false branch. Now the value of id is at least 0.

if (flipNegative && id < 0)
{
    assignment: Assigning: id = -id.

    id = -id;
}

int mapped = _mapping.MappedID(id, lenient);

at_least: At condition id < 0, the value of id must be at least 0.
dead_error_condition: The condition id < 0 cannot be true.

if (flipNegative && id < 0)
{
    CID 14128 (#1 of 1): Logically dead code (DEADCODE)

    dead_error_line: Execution cannot reach this statement mapped = -mapped;

    mapped = -mapped;
}

_target.WriteInt(mapped);
return mapped;
A presentation ends with the conclusion, which is your last chance to sum up the information you’re trying to convey.

So as you see, a presentation has three parts.

And two of them are useless.

If you’ve been listening, I’ve demonstrated that that is not true.

I agree. You’ve shown that all three parts are useless.

© 2010: Scott Meyer

Special thanks to Scott at BasicInstructions.net
Summing up

• Even good languages make it too easy to write bugs
• Bugs are easy to cause and hard to find, even for experts
• The earlier you find the bug, the cheaper it is to fix
• Static analysis tools do a lexical, grammatical, semantic, interprocedural control flow and intra procedural control flow analysis to find specific defect patterns...
• ... and many heuristics to eliminate false positives
• Writing good defect presentations is a hard problem
• Use of static analysis tools is growing in industry, but are not yet considered must-haves
• The stuff you're learning is actually useful!
More information

• Learn about Coverity at www.Coverity.com
• Read “A Few Billion Lines Of Code Later”
• Find me on Twitter at @ericlippert
• Or read my C# blog at www.EricLippert.com
• Or ask me about C# at www.StackOverflow.com
if (defaultValue > maxValue)
    _defaultValue = maxValue;
else if (defaultValue < minValue)
    _defaultValue = minValue;
else
    _defaultValue = defaultValue;
if (value > maxValue)
    _value = maxValue;
else if (defaultValue < minValue)
    _value = minValue;
else
    _value = value;
original: `defaultValue < minValue` looks like the original copy.

```cpp
    else if (defaultValue < minValue)
    {
        _defaultValue = minValue;
    }
    else
    {
        _defaultValue = defaultValue;
    }
    _manual = manual;
    if (value > maxValue)
    {
        _value = maxValue;
    }
```

CID 19155 (#1 of 1): Copy-paste error (COPY_PASTE_ERROR)

`copy_paste_error: defaultValue in defaultValue < minValue` looks like a copy-paste error.

Should it say `value` instead?

```cpp
    else if (defaultValue < minValue)
    {
        _value = minValue;
    }
    else
    {
        _value = value;
    }
```