What’s it all about?

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken
Overview

We’re going to make a quick tour through the course material today:

- Understanding Failure Circumstances
- Examining the Run
- Isolating the Defect
- Delta Debugging
- Program Slicing
- Detecting Anomalies

This is just to provide an overview—if you don’t get it at first, just sit back and relax :-)
A Simple Example

The sample program is supposed to sort its arguments:

```
$ sample -7 14 5 -4 1 2 3
Output: -7 -4 1 2 3 5 14
$ _
```

Unfortunately, sample has a defect:

```
$ sample -11 14 7 5 4 1 2 3
Output: -11 0 1 2 3 4 5 7
$ _
```

This will be our ongoing example today.
From Defects to Failures

A failure comes to be in three stages:

\[ \text{Defect} \rightarrow \text{Infection} \rightarrow \text{Failure} \]

A defect is an *error in the program* (code).

An infection is an *error in the program state*.

A failure is an observable *error in the program behavior*.

The issue of debugging is to

- relate an observed failure to a defect and
- to remove the defect such that the failure no longer occurs.
From Defects to Failures (2)

Not every defect causes an infection, and not every infection causes a failure.

This is the curse of testing:

*Testing can only show the presence of defects, but never their absence.* (Dijkstra)

On the other hand, each failure can be traced back to some infection, and each infection can be traced back to a defect.

This is what we shall do with the sample program.
Understanding Failure Circumstances

Before you rush to your favorite debugger, first try to understand under which circumstances the failure occurs. The idea is to identify the relevant circumstances.
Simplifying sample input

We try to simplify the input

```
$ sample -11 14 7 5 4 1 2 3
Output: -11 0 1 2 3 4 5 7
$ sample -11 14 7 5
Output: -11 0 5 7
$ sample -11 14
Output: -11 0
$ sample -11
Output: -11
$ 
```

This input simplification is a general pattern which isolates a failure cause (here: the input 14).
Debugging Patterns

Idea: A set of *patterns* where each pattern solves a specific debugging problem.
**Pattern: The Scientific Method**

1. Observe a failure.

2. Invent a *hypothesis* as to the failure cause that is consistent with the observations and the necessary conditions.

3. Use the hypothesis to make predictions.

4. Test the hypothesis by experiments or further observations and modify the hypothesis in the light of your results.

5. Repeat steps 3 and 4 until you found the actual cause.

<table>
<thead>
<tr>
<th>#</th>
<th>Run with args...</th>
<th>Outcome</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8 -7 5 -4 1 2 3</td>
<td>✔️</td>
<td>original pass</td>
</tr>
<tr>
<td>2</td>
<td>-11 14 7 5 4 1 2 3</td>
<td>❌—output contains 0</td>
<td>original failure</td>
</tr>
<tr>
<td>3</td>
<td>-11 14 7 5</td>
<td>❌—output contains 0</td>
<td>simplified</td>
</tr>
<tr>
<td>4</td>
<td>-11 14</td>
<td>❌—output contains 0</td>
<td>simplified</td>
</tr>
<tr>
<td>5</td>
<td>-11</td>
<td>✔️</td>
<td>isolated difference</td>
</tr>
</tbody>
</table>

Conclusion: extra 14 is failure-inducing
Understanding Infection Origins

Program states
Variable and input values
Program execution

Sane state
Erroneous code
Infected state
Observer sees failure
Sane state

Erroneous code
Infected state
Observer sees failure
Debugging as a search problem

**Search in space.** Each single state is composed of thousands or even millions of variables. Debugging means to separate the infected variables from the sane variables.

**Search in time.** A program execution consists of thousands, millions or even billions of states. Debugging means to isolate the infection—the transition from a sane state towards an infected state.

Fortunately, debugging is not *that* difficult:
Good programming style limits the information flow between units (= functions, modules, objects...)

*A divided state is much easier to conquer!*
int main(int argc, char *argv[]) {
    int *a;
    int i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    printf("Output: ");
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);

    return 0;
}
static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}
**Isolating the infection**

The 0 in the output stems from a[1].

(See printf)

a[1] stems from shell_sort.

(See shell_sort)

At shell_sort invocation, the state is *infected*.

\[
\begin{align*}
  a[0] &= -11 \\
  a[1] &= 14 \\
  a[2] &= 0 \\
  size &= 3
\end{align*}
\]

The infection occurs at the invocation of shell_sort:

```c
shell_sort(a, argc);
```
How the Failure came to Be

1. The array a[] is allocated and initialized with the correct number of elements (2).
2. shell_sort is invoked such that the size parameter is 3.
3. This causes shell_sort to access a[] beyond the allocated space—namely at a[2].
5. During the sort, a[2] is eventually swapped with a[1].
6. Thus the zero value of a[1] is printed, causing the failure.
Fixing the Defect

We replace

    shell_sort(a, argc);

by the correct invocation

    shell_sort(a, argc - 1);

Now we must repeat the test:

    $ sample -11 14
    Output: -11 14
    $ 

The sample program is fixed.
Our Debugging Process

**Simplify Input.** Reducing the number of sample arguments.

**Isolate Origins.** Relating a[1] back to shell_sort.

**Log Execution.** Looking into the values at shell_sort.

**Isolate Infection.** Finding out that the state was infected at the invocation of shell_sort.

**Most General Fix.** We’re done!
The General Debugging Process

1. Reproduce the failure, using **REPRODUCE PROBLEM**.

2. Use the **SCIENTIFIC METHOD** to
   - isolate failure-inducing circumstances
   - trace back infected values to its origins
   - isolate the moment of infection

3. Fix the program, using **MOST GENERAL FIX**.

This is a pattern, namely **DEBUG A PROGRAM**
Isolating Relevant Circumstances

We have

- A means to *simplify* the input (i.e. the arguments)
- A *testing function* that executes the program and tells us whether the failure occurs

Idea: *Automated simplification*

While we can simplify the input such that the failure still occurs, do it.

This is a pattern: **SIMP queens INPUT!**
Isolating Relevant Circumstances (2) 

The pattern **DE**LT**A** DEBUGGING automates simplifying the input

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1000 lines</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>500 lines</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>250 lines</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>125 lines</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>125 lines</td>
<td>X</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>63 lines</td>
<td>✓</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>40 characters</td>
<td>X</td>
</tr>
<tr>
<td>58</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>20 characters</td>
<td>✓</td>
</tr>
<tr>
<td>59</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>20 characters</td>
<td>✓</td>
</tr>
<tr>
<td>60</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>30 characters</td>
<td>✓</td>
</tr>
<tr>
<td>61</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>20 characters</td>
<td>X</td>
</tr>
<tr>
<td>62</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>10 characters</td>
<td>X</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>8 characters</td>
<td>✓</td>
</tr>
<tr>
<td>76</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>8 characters</td>
<td>✓</td>
</tr>
<tr>
<td>77</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>8 characters</td>
<td>✓</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td><code>&lt;SELECT NAME=&quot;priority&quot; MULTIPLE SIZE=7&gt;</code></td>
<td>8 characters</td>
<td>X</td>
</tr>
</tbody>
</table>
Isolating Relevant Circumstances (3)

Other relevant circumstances that can be isolated include:

**Input differences:**
The character `<` in the input causes Mozilla to fail.

**Schedule differences:**
The failure occurs if the thread switch occurs *here*.

**Version differences:**
The change in *that* line caused the failure.
Program Slicing

While debugging sample, we traced back a[1] to its origins. This can also be partially automated, using PROGRAM SLICING.

Basic idea of slicing: isolate dependencies of variable values by analyzing the program code.

Any value of a[i] at the program end, for instance, is dependent on

- a[i] = atoi(argv[i + 1]);
- a[j] = v;
- a[j] = a[j - h];

a[i] can get its value only from these locations!
**Dynamic Slicing**

*Dynamic* slicing tracks the dependencies for a *concrete program run*.

Using a dynamic slicer, we can determine that `a[1]` was last assigned in

\[
a[j] = v; \quad // \quad j = 1, \; v = 0
\]

The previous assignment to `v` was at

\[
v = a[i]; \quad // \quad i = 2
\]

We have traced back that `a[1]`’s bad value came from `a[2]`. But `i` is 2 because of size, so `a[1]` is also dependent on size.
**Cause-Effect Chains**

Basic idea: **COMPARE STATES** of a *passing run* and a *failing run*; differences indicate failure causes.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>✔️</td>
<td>2</td>
<td>&quot;sample&quot;</td>
<td>&quot;-11&quot;</td>
<td>NULL &quot;14&quot;</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>8</td>
<td>✔️</td>
<td>2</td>
<td>&quot;sample&quot;</td>
<td>&quot;-11&quot;</td>
<td>NULL &quot;14&quot;</td>
<td>-11</td>
<td>256</td>
<td>512</td>
<td>2</td>
</tr>
<tr>
<td>37</td>
<td>✔️</td>
<td>2</td>
<td>&quot;sample&quot;</td>
<td>&quot;-11&quot;</td>
<td>NULL &quot;14&quot;</td>
<td>-11</td>
<td>256</td>
<td>512</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Which one of these differences is relevant for the failure?
Cause-Effect Chains (2)

Basic idea: We one difference at a time and check the outcome.

1. If we apply no difference at all, a[1] is eventually 256; the output is -11. The program passes.

2. If we run the program until Line 8, set argc from 2 to 3 and resume execution, a[1] is still 256; the output is -11 256.

3. If we repeat the experiment and also set argv[2] from NULL to "14", a[1] and output are unchanged.

4. If we also set a[1] from 256 to 14, variable a[1] remains 14; the output is -11 14.

5. If we also set a[2] from 512 to 0, variable a[1] remains 14, the output is still -11 14.

6. If we also set size from 2 to 3, variable a[1] becomes 0; the output is -11 0. Only with this last step did the failure occur—a[1] became zero.
Cause-Effect Chains (3)

Your request has been successfully processed.

Diagnosis

This is what happens in your program when it is invoked as "sample -11 14". (More info...)

1. Execution reaches sample.c, line 35.
   Since the program was invoked as "sample -11 14", local variable a[2] is now 0.

2. Execution reaches sample.c, line 9.
   Since a[2] was 0, variable a[2] in stack frame #1 is now 0.

3. Execution ends.
   Since a[2] was 0, the output now contains the word "0". The program fails. (Edit Symptoms)

Need more details? Check the effects you want to focus upon and Re-debug it!
Isolating Infections

Issue: Find the moment in time when the state changes from **sane** to **infected**.

Automation requires means to *check the sanity of the state*.
Isolating Infections (2)

Assertions.

\[
\text{assert(is\_sorted(a, size));}
\]
\[
\text{assert(is\_permutation\_of(a, original\_a, size));}
\]

Checking the heap.

\[
\text{
$\text{MALLOCCHECK=2 sample -11 14}$
Output: -11 14
$\text{ }$
}\]

Checking array boundaries.

\[
\text{
$\text{gcc -g -o sample-with-efence sample.c -lefence}$
$\text{sample-with-efence -11 14}$
Electric Fence 2.1
Segmentation fault (core dumped)
$\text{ }$
}\]
Searching for Anomalies

Another place to search infections at are *anomalies*.

An *anomaly* in a program run is a property which deviates from the (non-failing or *normal*) standard.

We can look for anomalies

- *in the code*—**CHECK STYLE**
- *in the execution*—**COMPARE COVERAGE**
- *in the data*—**DETECT INVARIANTS**
**Anomaly in the Execution**

We compare the *coverage* of the passing and the failing run:

<table>
<thead>
<tr>
<th>Coverage</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️ ×</td>
<td>static void shell_sort(int a[], int size)</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>{</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>int i, j;</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>int h = 1;</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>do {</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>h = h * 3 + 1;</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>} while (h &lt;= size);</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>do {</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>h /= 3;</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>for (i = h; i &lt; size; i++)</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>{</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>int v = a[i];</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>for (j = i; j &gt;= h &amp;&amp; a[j - h] &gt; v; j -= h)</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>a[j] = a[j - h];</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>if (i != j)</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>a[j] = v;</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>}</td>
</tr>
<tr>
<td>✔️ ×</td>
<td>} while (h != 1);</td>
</tr>
<tr>
<td>✔️</td>
<td>}</td>
</tr>
</tbody>
</table>
Anomaly in the Data

Another area to search anomalies for is *data*.

**Detect Invariants**—see how the program data differs from *inferred invariants*:

**At start of shell-sort**

```
size == size(a[])
a[] one of  [-11, 14], [7, -1, 25, 9], [14, 11]
size one of  2, 4
```

**At exit of shell-sort**

```
orig(size) == orig(size(a[]))
a[] one of  [-11, 14], [-1, 7, 9, 25], [11, 14]
```
Invariants in the Data (2)

At start of `main`

\[
\begin{align*}
\text{argc} &= \text{size}(\text{argv}[]) - 1 \\
\text{argc} \text{ one of } 3, 5 \\
\text{size}(\text{argv}[]) \text{ one of } 4, 6 \\
\text{argv}[\text{argc}..] &= \text{[null]} \\
\text{argv}[\text{argc}..] \text{ elements} &= \text{null} \\
\text{argv}[\text{argc}+1..] &= \text{[]} \\
\end{align*}
\]

At exit of `main`

\[
\begin{align*}
\text{argv}[] &= \text{orig}(\text{argv}[]) \\
\text{return} &= 0 \\
\text{argv}[\text{orig}(\text{argc})..] &= \text{[null]} \\
\text{argv}[\text{orig}(\text{argc})..] \text{ elements} &= \text{null} \\
\text{argv}[\text{orig}(\text{argc})+1..] &= \text{[]} \\
\end{align*}
\]
Debugging Details

More material to be covered:

- *Reproducing* the Problem (faithfully and automatically)
- *Fixing* the Program (in the best possible way)
- *Preventing* Failures

... all in the remaining course.
Concepts

In general, a failure comes to be in three stages:

1. The programmer creates a defect in the program code (also known as bug or fault).
2. The defect causes an infection in the program state.
3. The infection causes a failure—an externally observable error.

Not every defect results in an infection, and not every infection results in a failure. Yet, every failure can be traced back to some infection, which again can be traced back to a defect.
Concepts (2)

- **Debugging a program** consists of three activities:
  1. *Reproducing* the failure,
  2. *Relating* the failure to a defect in the program, and
  3. *Fixing* the defect.

Of these three activities, the second is by far the most time-consuming.

- **Debugging patterns** encapsulate solutions to specific debugging problems.

- A variety of systematic and automated approaches is available that help in debugging.

See remaining course!